Technická univerzita v Liberci

Fakulta mechatroniky a mezioborových inženýrských studií

Studijní program: N 2612 – Elektrotechnika a informatika

Studijní obor: 1802T007 – Informační technologie

Oprava překlepů dotazů zadávaných do vyhledávače

Spell correction of web search queries

Diplomová práce

Autor: **Bc. Stanislav Nowak**

Vedoucí diplomové práce: Mgr. Jiří Vraný

Konzultant: Ing. Radim Škrob

V Liberci dne 16. 5. 2008

Zadani

Prohlášení

Byl jsem seznámen(a) s tím, že na mou diplomovou práci se plně vztahuje zákon č. 121/2000 o právu autorském, zejména § 60 (školní dílo).

Beru na vědomí, že TUL má právo na uzavření licenční smlouvy o užití mé diplomové práce a prohlašuji, že **souhlasím** s případným užitím mé diplomové práce (prodej, zapůjčení apod.).

Jsem si vědom(a) toho, že užít své diplomové práce či poskytnout licenci k jejímu využití mohu jen se souhlasem TUL, která má právo ode mně požadovat přiměřený příspěvek na úhradu nákladů, vynaložených univerzitou na vytvoření díla (až do její skutečné výše).

Diplomovou práci jsem vypracoval(a) samostatně s použitím uvedené literatury a na základe konzultací s vedoucím bakalářské práce a konzultantem.

Datum

Podpis

Poděkování

Na tomto místě bych chtěl poděkovat zejména vedoucímu své práce Mgr. Jiřímu Vranému. Rovněž bych rád poděkoval i Ing. Štěpánu Šrobovi a Mgr. Radimu Řehůrkovi ze společnosti Seznam.cz za poskytnuté podklady, bez kterých by diplomová práce nemohla vzniknout.

Abstrakt

Cílem práce bylo vytvořit korektor překlepů uživatelských dotazů zadávaných do internetového vyhledávače. Korektory překlepů v dotazech se využívají především pro zvýšení uživatelského komfortu při používání vyhledávače. Za svou současnou popularitu vděčí především vyhledávači společnosti Google. Dnes již patří korektor dotazů mezi standardní funkce a setkáme se s ním i na většině českých vyhledávačů.

Korektor překlepů pro vyhledávač se od těch, které známe například z textových procesorů, liší v několika ohledech. Především jsou na něj kladeny podstatně vyšší výkonnostní nároky a rovněž se musí umět vypořádat se specifickým druhem jazyka dotazů zadávaných do vyhledávače. Diplomová práce staví na poznatcích statistické lingvistiky a byla řešena na základě bayesovského přístupu. Korektor se tak rozdělil na dvě samostatné části, a to jazykový a chybový model. Pro jazykové modelování byl využit stochastický n-gramový model. Chybový model je založen na principu minimální editační vzdálenosti a nejpravděpodobnějšího zarovnání řetězců.

K trénovaní obou modelů byla použita data sestavená ze záznamů uživatelských dotazů zadávaných do vyhledávače společnosti Seznam.cz. Při testování dosáhl implementovaný korektor uspokojivých výsledků, přesto však zůstává prostor pro mnohá vylepšení.

Klíčová slova: oprava překlepů, statistická lingvistika, zpracování přirozeného jazyka, vyhledávač

Abstract

The goal of this diploma thesis was to develop a spelling correction program of web search queries. The spelling correction of search queries is used to provide a richer user experience. Its current popularity was gained thanks to Google search engine. Nowadays the spelling correction became a standard feature of many search engines and we can see it at Czech search engines too.

Search query correction differs from ones that we use for example in text processor in several ways. It must be able to deal with high performance requirements and also with a specific type of language of web search queries.

The diploma thesis is based on computational linguistics and Bayesian approach. By using Bayesian method we get two separated components: language and error model. For the language model we use stochastic n-gram based model. The error model is based on minimal string distance and maximum probability alignment.

For training of both models were used data build from search queries logs of Seznam.cz search engine. Testing shows satisfactory results however there are still several improvements that can be made.

Keywords: spell correction, computational linguistics, NLP, search engine

Obsah

1 Úvod

Přestože výzkumné práce v oblasti automatických oprav textu započaly již začátkem 60. let dvacátého století, dodnes je toto téma živé a věnuje se mu velké úsilí a pozornost. Bylo dosaženo dílčích úspěchů a v současnosti máme k dispozici několik velmi kvalitních akademických i komerčních korektorů. Stále však zůstává prostor pro zlepšení, zejména v oblasti úspěšnosti oprav a schopnosti vypořádat se s obecným tématem. Důležitou hnací silou pro další vývoj je i zvyšování výkonu výpočetních prostředků. Už v šedesátých letech byla známa řada algoritmů, které nemohly být implementovány z důvodu vysoké výpočetní náročnosti a dostalo se na ně až relativně nedávno.

Kromě běžných textů se objevují nové výzvy vyžadující specifický přístup. Takovou úlohou je i oprava překlepů uživatelských dotazů zadávaných do internetového vyhledávače s využitím kontextu. Na korektor dotazů pro vyhledávač jsou kladeny velké výkonností nároky a musí účinně pracovat pod velkou zátěží. S tímto požadavkem se dnes dokážeme relativně snadno vypořádat použitím optimalizovaných algoritmů, speciálních datových struktur nebo zvýšením výkonu výpočetních prostředků.

Závažnějším problémem zůstává jazyk dotazů vkládaných do vyhledávače. Jazyková struktura dotazu se zásadně liší od stavby běžné české věty. Dotaz se v průměru skládá ze tří slov, a je tedy o poznání kratší než průměrná česká věta. Mezi slovními druhy převažují podstatná jména, přídavná jména a příslovce, zatímco počet sloves je výrazně nižší. Charakteristický je i velký výskyt cizojazyčných výrazů, převážně anglických. V zadávaných dotazech se často objevují názvy obchodních značek a výrobků, jejich modelových a typových řad, nebo dokonce celé www adresy. Frekvence výskytů překlepů je rovněž vyšší než v běžném textu.

V historii automatických oprav se objevilo několik cest k řešení zadané úlohy. Jako nejúčinnější se ukázal přístup postavený na statistických metodách. V současnosti se s jinými postupy setkáme jen výjimečně. Pro tvorbu korektoru byl zvolen bayesovský přístup. Ten nám umožnil rozdělit úlohu na dvě samostatné části, které mohou být trénovány a testovány nezávisle. Dvě zmíněné části jsou: jazykový a chybový model.

Jedním z požadavků na korektor bylo, aby uměl vyžít kontextu okolních slov dotazu. Toho jsme dosáhli použitím $n-$gramového modelu jako jazykového modelu.

Pro tvorbu chybového modelu byl zvolen princip založený na minimální editační vzdálenosti a nejpravděpodobnějšího zarovnání řetězců. Předností tohoto přístupu je, že pomocí editačních operací přirozeným způsobem popisuje uživatelské překlepy v dotazech.

Při přípravě podkladů se nepodařilo zajistit žádný český zdroj zaměřený přímo na opravu překlepů. V řadě prací byly popsány jen částečné informace využitelné pro tvorbu korektoru, například jazykové modelovaní. V kontrastu jsou anglické zdroje, ze kterých se podařilo získat velké množství použitelných podkladů a informací. Z toho důvodu se diplomová práce snaží být uceleným česky psaným shrnutím v oblasti korekce překlepů. V práci jsou proto nejdříve podrobně rozebrány obecné principy opravy překlepů a na ně navazuje popis specifických řešení pro vyhledávač.

Úloha korekce překlepů spadá do oblasti statistické lingvistiky, proto je první kapitola věnována právě lingvistice. Krátce si ji popíšeme, zaměříme se především na matematickou lingvistiku a představíme si její základní disciplíny.

V druhé kapitole se seznámíme s teoretickým aparátem, na kterém je vybudován korektor překlepů. Postupně budeme procházet jednotlivé části, podrobně rozebereme jejich teoretické zázemí a nakonec vše spojíme dohromady a získáme tak korekční algoritmus. V kapitole jsou zmíněny a krátce popsány i další možné přístupy k opravě překlepů.

Součástí zadání práce bylo vytvoření funkčního prototypu korektoru. Jeho implementace je popsána v praktické části, která slouží zároveň jako dokumentace k programu. Dále se zmíníme o přípravě a tvorbě jazykového a chybového modelu.

V předposlední kapitole ověříme kvalitu návrhu a implementace korektoru na rozmanitých úlohách. Zaměříme se hlavně na úspěšnost za použití různě velkých jazykových modelů a nastavení úrovní přesnosti chybového modelu.

Na závěr shrneme a okomentujeme výsledky testování. Rovněž se zmíníme o možných zlepšeních a doporučíme směr dalšího vývoje.

2 Lingvistika

Důležitým zdrojem poznatků použitých v diplomové práci se stala právě lingvistika, proto se o ní krátce rozepíšeme a zaměříme se na matematickou lingvistiku. Informace pro tuto kapitolu byly čerpány především z [1].

Lingvistika neboli jazykověda (lingua je latinsky jazyk) je věda o jazycích, jejich třídění, stavbě a zvukové i psané podobě. Jako samostatná věda vznikla začátkem 19. století. Její zásadní průlom nastal až ve druhé polovině 20. století, kdy došlo ve vědě k výrazné změně a začal se klást velký důraz na pomezní disciplíny.

Jestliže se v dřívějších obdobích lingvisté zabývali buď systémem jazyka (strukturalismus), nebo některou z jeho částí (srovnávací a historická gramatika), pak v druhé polovině 20. století převážil zájem o takové disciplíny, v nichž se s úspěchem kombinují otázky lingvistiky s tématy a metodami, jako je matematika, logika, psychologie, neurologie, sociologie atd.

Vznikla tak řada pomezních disciplín, které v současné lingvistice získaly dominantní postavení a dnes stojí v samém centru jejího zájmu. Řadíme do nich zejména matematickou lingvistiku, psycholingvistiku, sociolingvistiku a mnohé další.

2.1 Matematická lingvistika

Termínem matematická lingvistika se dnes označují takové disciplíny, v nichž se při výzkumu jazyka používají matematické metody. Protože je aplikace matematických metod mnohostranná, rozlišují se v rámci matematické lingvistiky tři dílčí disciplíny – lingvistika kvantitativní, algebraická a počítačová. První dvě jsou převážně teoretické, třetí aplikuje jejich poznatky v praxi, zejména při počítačovém překladu.

2.1.1 Kvantitativní (statistická) lingvistika

Kvantitativní lingvistika popisuje jazyk pomocí statistických metod. Tato disciplína má delší tradici než algebraická a počítačová, protože statistické metody se v lingvistice objevovaly už od konce 19. století. Řada lingvistů si všimla, že jednotlivé jazykové jevy (hlásky, písmena, slabiky, gramatické kategorie, slovní spojení atd.) se vyskytují s různou nerovnoměrnou frekvencí. Znalosti frekvencí jednotlivých jevů umožnily použití teorie pravděpodobnosti k popisu jazyka.

Statistické metody dostaly v padesátých letech 20. století nový impuls v podobě teorie komunikace a informace, které do lingvistiky zavedly takové pojmy, jako například

entropie, perplexita, redundance, bit a šum.

Většina poznatků a metod použitých v diplomové práci pochází právě z této disciplíny.

2.1.2 Algebraická lingvistika

Algebraická lingvistika je název pro jazykové teorie opírající se o jiné matematické metody, než jsou kvantitativní. Převážně se jedná o metody algebraické a logické, které přistupují k jazyku jako k formálnímu modelu.

Reprezentativním příkladem je transformační a generativní gramatika Noama Chomského, která představuje jeden z nejpropracovanějších modelů jazyka. S úspěchem se aplikuje obzvláště na jazyky s pevným slovosledem a chudou morfologií, jako je angličtina.

2.1.3 Počítačová (strojová) lingvistika

Neobyčejný rozvoj počítačů v druhé polovině 20. století způsobil značné změny v metodách vědeckého výzkumu v nejrůznějších disciplínách, mezi jinými také v lingvistice. Pro celou řadu operací lingvistického charakteru, které se provádějí na počítačích, se používá souhrnný název – počítačová lingvistika. Ta v praxi využívá poznatky kvantitativní a algebraické lingvistiky a zpětně ovlivňuje rozvoj obou disciplín.

Nejznámější činností strojové lingvistiky je strojový překlad – jeho cílem je převést text výchozího jazyka na jazyk cílový. Kromě strojového překladu se počítače v lingvistice s úspěchem uplatňují i při jiných činnostech, jako je např. spektrální analýza mluvené řeči, zpracování frekvenčních seznamů, rozpoznání psaného slova nebo oprava překlepů.

Korpusová lingvistika

Jedná se o mladé odvětví, v současnosti spadající do oboru počítačové lingvistiky, jehož rozvoj je spjat s příchodem počítačů a informačních technologií. Tato disciplína zkoumá jazyk pomocí elektronických jazykových korpusů a zabývá se i výstavbou těchto korpusů, jejich zpracováním a příslušnou metodologií. V současnosti se korpusem rozumí rozsáhlý vnitřně strukturovaný a ucelený soubor textů daného jazyka, elektronicky uložený a zpracovávaný.

3 Teoretický rozbor

V této kapitole se zaměříme na teoretický aparát, který stojí za opravou překlepů. Budeme se opírat především o statistické metody a hned v úvodu sjednotíme terminologii. Užitím bayesovského přístupu rozdělíme úlohu na dvě samostatné části. Jazykový i chybový model důkladně popíšeme a uvedeme problémy spojené s jejich tvorbou. Na závěr obě části opět spojíme a získáme korekční algoritmus. V krátkosti se budeme věnovat i dalším rozšířeným korekčním algoritmům a metodám.

3.1 Přehled statistických pojmů

Krátký exkurs do statistiky slouží především k sjednocení terminologie a notace.

Deterministický děj pozorování nebo pokus, který má v daných podmínkách jednoznačný výsledek

Stochastický (náhodný) děj pozorování nebo pokus, který může v daných podmínkách vést k různým výsledkům

Náhodný pokus stochastický děj, který je za týchž podmínek nekonečně opakovatelný

Náhodný jev A výsledek náhodného pokusu, o němž lze jednoznačně říci, že nastal, nebo nenastal

Statistická definice pravděpodobnosti Opakujme náhodný pokus $N-$krát, přičemž předpokládejme, že výskyt náhodného jevu A pozorujeme v K případech. Číslo K se nazývá četností jevu A. Poměr se pak označuje jako poměrná či relativní četnost jevu A. Jestliže se s rostoucím N, tedy se zvyšováním počtu opakování pokusu, relativní četnost blíží nějakému číslu, pak toto číslo můžeme považovat za pravděpodobnost daného jevu.

$$P(A) = \lim_{N\to\infty} \frac{K}{N} \tag{1}$$

Sdružená pravděpodobnost označuje pravděpodobnost, že náhodné jevy A a B nastanou současně

$$P(A|B) = P(A, B) = P(AB) \tag{2}$$

Podmíněná pravděpodobnost označuje pravděpodobnost, zda se jev A může vyskytnout pouze tehdy, vyskytl-li se jev B, jehož pravděpodobnost je $P(B) > 0$

$$P(A|B) = \frac{P(A \cap B)}{P(B)} \tag{3}$$

Bayesův vzorec udává, jakým způsobem vypočítáme pravděpodobnosti $P(A|B)$ jevu A za podmínky, že nastal jev B, jestliže známe apriorní pravděpodobnosti $P(A)$ a podmíněnou pravděpodobnost $P(B|A)$. Bayesův vzorec má tvar:

$$P(A|B) = \frac{P(B|A)\,P(A)}{P(B)}. \tag{4}$$

3.2 Bayesovský přístup k opravě překlepů

Při řešení úlohy opravy překlepů využíváme statistické metody. Předpokládejme, že $W = \{w_1, w_2, w_3, \ldots, w_N\}$ je posloupnost N slov představující dotaz zadaný do vyhledávače. Dále nechť je $C = \{c_1, c_2, c_3, \ldots, c_N\}$ opravená posloupnost slov.

Cílem je nalézt nejpravděpodobnější opravenou posloupnost slov C pro dotaz W:

$$\hat{C} = argmax_C P(C|W), \tag{5}$$

kde $P(C|W)$ je podmíněná pravděpodobnost, že C je oprava dotazu W a funkce $argmax_C$ v tomto vztahu znamená nalezení posloupnosti C takové, pro kterou je $P(C|W)$ maximální. Uvedený vztah však nedokážeme nikterak ohodnotit. Například pokud máme chybné slovo *porlední*, z uvedeného vztahu nedovedeme určit, zda je správnější oprava *poslední* nebo *polední*.

V případě, že použijeme Bayesovo pravidlo, platí:

$$\hat{C} = argmax_C \frac{P(C)P(W|C)}{P(W)}, \tag{6}$$

kde $P(C)$ je apriorní pravděpodobnost posloupnosti slov C, tedy pravděpodobnost, s jakou uživatel zadá do vyhledávače pravděpodobnost slov C. $P(W|C)$ vyjadřuje pravděpodobnost, že byla zadána chybná posloupnost W, pokud měla být správně zadána posloupnost C. $P(W)$ je apriorní pravděpodobnost chyby a vzhledem k tomu, že je konstantní, můžeme ji při hledání maxima ignorovat. Výsledná rovnice má tedy tvar:

$$\hat{C} = argmax_C P(C)P(W|C). \tag{7}$$

Z tohoto vztahu vyplývá, že problém nalezení nejlepší opravy C k zadané posloupnosti W lze řešit pomocí dvou oddělených pravděpodobností $P(C)$ a $P(W|C)$, které mohou být modelovány a trénovány nezávisle na sobě. Apriorní pravděpodobnost $P(C)$ nese informaci o jazykovém modelu a podmíněná pravděpodobnost $P(W|C)$ o chybovém modelu.

Z uvedeného výkladu vyplývá, že úloha opravy překlepu může být tedy rozdělena do tří dílčích úloh:

1. vytvoření jazykového modelu $P(C)$
2. vytvoření chybového modelu $P(W|C)$
3. nalezení nejpravděpodobnější posloupnosti slov

3.3 Jazykový model

Účelem jazykového modelu je nalézt určitá pravidla a stanovit taková omezení, pomocí nichž můžeme v modelovaném jazyce ze slov sestavit větu. K modelování jazyka můžeme přistupovat dvěma způsoby a to deterministickým a stochastickým. V řešené úloze se budeme držet výhradně stochastického tedy pravděpodobnostního přístupu.

Pomocí stochastického jazykového modelu stanovíme pro každou posloupnost slov $W = \{w_1, w_2, w_3, \ldots, w_k\}$ apriorní pravděpodobnost $P(W)$, kterou zjistíme ze vztahu pomocí řetězového pravidla:

$$\begin{aligned} P(W) &= P(w_1^k) = P(w_1, w_2, w_3, \ldots, w_k) \\ &= P(w_1)P(w_2|w_1)P(w_3|w_1, w_2)\ldots P(w_n|w_1, w_2, \ldots, w_{k-1}) \\ &= \prod_{i=1}^{k} P(w_i|w_1, w_2, \ldots, w_{i-1}). \end{aligned} \tag{8}$$

V uvedeném rozkladu pravděpodobnosti $P(W)$ jsou podmíněné pravděpodobnosti výskytu slova w_i podmíněny pouze svou historií, tj. posloupností slov $w_1 \ldots w_{i-2}w_{i-1}$.

3.3.1 n-gramové modely

Aby bylo možné vytvořit jazykový model podle vztahu 8, museli bychom určit apriorní pravděpodobnost $P(w_1^k)$ všech možných posloupností slov do délky k. Tyto posloupnosti je však velmi obtížné určit a téměř nemožné ohodnotit. Podle Markovova předpokladu, který říká, že blízkou budoucnost lze odhadnou z krátkodobé historie, budeme jazykový model aproximovat Markovovým modelem $(n-1)-$vého řádu. Tyto modely se nazývají $n-$gramové. Termínem $n-$gram se rozumí posloupnost n za sebou jdoucích slov získaných například z trénovacího korpusu. $n-$gramy s $n = 0$ nazýváme zerogramy a $n-$gramy s $n = 1$ unigramy. Mezi používanější $n-$gramy patří bigramy (kde $n = 2$) a trigramy ($n = 3$). Intuice nám říká, že čím bude n větší, tím lépe bude aproximován daný jazykový model. Z praktických důvodů se za n nejčastěji volí 2 (bigramový model) nebo 3 (trigramový model).

V $n-$gramovém modelu je podmíněná pravděpodobnost $P(w_k|w_1, w_2, \ldots, w_{k-1})$ slova w_k závislá pouze na $n-1$ předchozích slovech a aproximuje se vztahem:

$$P(w_k|w_1, w_2, \ldots, w_{k-1}) \approx P(w_k|w_{k-n+1}, w_2, \ldots, w_{k-1}). \tag{9}$$

Pro $P(w_1^k)$ tady platí:

$$P(w_1^k) \approx \prod_{i=1}^{k} P(w_i|w_{i-n+1}, w_2, \ldots, w_{i-1}). \tag{10}$$

Jazykové $n-$gramové modely jsou vhodné především pro jazyky s relativně pevným pořadím slov ve větě, neboť zde existují silné statistické závislosti mezi výskyty za sebou následujících slov.

Mezi další přednosti patří, že algoritmus výpočtu $n-$gramové statistiky je nezávislý na jazyku.

Ještě poznamenáme, že tvorba jazykového modelu pro češtinu je výrazně náročnější úloha než například pro angličtinu. Na vině je především velká morfologická bohatost češtiny a volný pořádek slov ve větě. Této problematice je věnována samostatná část.

V řešené úloze budeme používat unigramový a bigramový model.

3.3.2 Unigramový model

Unigramový model lze chápat jako frekvenční slovník jednotlivých slov v trénovacích datech. Z unigramového modelu se určuje apriorní pravděpodobnost jednotlivých slov. Pravděpodobnosti unigramů získáme ze vztahu:

$$P(w) = \frac{C(w)}{N}. \tag{11}$$

$C(w)$ je četnost slova w a N je celkový počet unigramů.

3.3.3 Bigramový model

Základní nevyhlazený bigramový jazykový model má podobu matice, kde řádky a sloupce jsou označeny slovy z unigramového modelu. Její prvky jsou podmíněné pravděpodobností určené pro všechny možné dvojice sousedních slov, které se objeví v trénovacích datech. Posloupnosti slov, které se v trénovacích datech neobjeví, mají hodnotu pravděpodobnosti rovnu nule. Takto vytvořený jazykový model je výchozí pro všechny vyhlazovací metody. Bigramový model používáme k uložení kontextu mezi slovy.

Podmíněné pravděpodobnos-ti prvků matice vyjádříme z četností získaných z trénovacích dat podle vztahu:

$$P(w_{n-1}w_n) = \frac{C(w_{n-1}w_n)}{C(w_{n-1})}, \tag{12}$$

nebo také jinak zapsáno:

$$P(w_n|w_{n-1}) = \frac{C(w_n|w_{n-1})}{C(w_{n-1})}, \tag{13}$$

kde $C(wn-1)$ je počet výskytů slova w_{n-1} a $C(w_{n-1}, w_n)$ je počet výskytů dvojic slov w_{n-1}, w_n.

$P(w_n|w_{n-1})$ hodnota vyjadřuje pravděpodobnost, že slovo w_n bude následovat po slově w_{n-1}. V naší úloze budeme potřebovat znát i pravděpodobnost, že slovo w_{n-1} předchází w_n. Využijeme vztahu:

$$P(w_{n-1}|w_n) = \frac{C(w_n|w_{n-1})}{C(w_n)}. \tag{14}$$

Prvky bigramové matice se nazývají parametry a jejich počet a struktura patří mezí důležité charakteristiky jazykového modelu.

Míru zastoupení nulových prvků matice budeme označovat jako řídkost.

3.3.4 Příprava, vlastnosti a vady jazykového modelu

Data, ze kterých se buduje jazykový model, se nazývají trénovací data nebo také trénovací korpus. Parametry jazykového modelu se ladí užitím heldout dat a účinnost modelu se ověřuje pomocí testovacích dat.

Trénovací korpus nebo data jsou v podstatě velmi rozsáhlý soubor textu. Z trénovacího korpusu získáme informace o frekvencích jednotlivých slov a jejich řetězců. Ze seznamu nejfrekventovanějších slov se sestavuje slovník a unigramový model. Z frekvencí dvojic slov se počítá bigramový jazykový model.

Příprava a výběr trénovacích dat má zásadní podíl na kvalitě jazykového modelu, a tedy i úspěšnosti opravy překlepů. Trénovací korpus se často sestavuje tematicky ušitý na míru dané úloze. Pokud se například řešená úloha zabývá medicínou, budeme korpus sestavovat z lékařských článků a skript a slovník bude obsahovat především lékařské termíny. Příliš úzce specializovaná trénovací data a omezený slovník způsobí, že jazykový model bude muset často řešit problém s neznámými slovy. O této vadě mluvíme jako o snížené citlivosti modelu.

V případě, že bude jazykový model obsahovat příliš rozsáhlý slovník, zvýší se počet jeho parametrů, řídkost a výpočetní náročnost řešené úlohy. Pokud nebudeme mít dostatečný objem dat pro trénování modelu s vysokým počtem parametrů, budou jejich hodnoty statisticky nevěrohodné. A naopak, nadměrně objemný korpus způsobí nežádoucí zvýšení výpočetní náročnosti úlohy.

Jak je vidět, při tvorbě jazykového modelu je nezbytné zvažovat mnoho navzájem se ovliňujících hledisek. Nastavení jazykového modelu má velký podíl na úspěšnosti řešené úlohy.

Ještě poznamenáme, že oprava překlepů není na řídkost trénovacích dat a nastavení parametrů tolik citlivá, jako například rozpoznávání mluveného nebo psaného slova [2].

3.3.5 Vyhlazování

Při tvorbě jazykového modelu vycházíme pouze z omezeného množství trénovacích dat a nedokážeme pokrýt všechna možná slova a jejich posloupnosti, které se mohou v modelovaném jazyce vyskytnout. V případě, že se daná posloupnost slov nebo

slovo v trénovacím korpusu neobjevily, uložíme na příslušnou pozici v $n-$gramové matice nulovou hodnotu. $n-$gramové matice jsou pak často řídké a obsahují velký počet nulových hodnot. Pokud by se v testovacím korpusu objevila posloupnost slov, pro kterou je v jazykovém modelu uvedena nulová hodnota, došlo by ke značnému zkreslení výsledků. Proto se využívají vyhlazovací algoritmy, jež nulovým hodnotám v matici přiřadí nenulové pravděpodobnosti. Vyhlazování se rovněž používá pro ladění parametrů jazykového modelu.

Je nutné zmínit, že vyhlazování způsobuje opačný problém a přiřadí nenulovou hodnotu i velkému množství $n-$gramů, které v modelovaném jazyce neexistují. I přes tento fakt bylo experimentálně ověřeno, že přesnost oprav se vyhlazením jazykového modelu výrazně zvýší [2].

Add-One

Add-One je nejjednodušší, ale ne příliš účinná metoda vyhlazování. Její princip spočívá v tom, že četnost všech $n-$gramů zvýšíme o jedna a následně spočítáme jejich pravděpodobnost.

Nejdříve si princip algoritmu předvedeme na vyhlazování unigramů. V je velikost slovníku, která odpovídá počtu různých unigramů. Pravděpodobnost pro vyhlazený unigram se vypočítá dle vztahu:

$$P_{+1}(w) = \frac{C(w) + 1}{N + V}. \tag{15}$$

Pro vyhlazený bigram platí:

$$P_{+1}(w_{n-1}w_n) = \frac{C(w_{n-1}w_n) + 1}{C(w_{n-1}) + V}. \tag{16}$$

Tento algoritmus neposkytuje příliš dobré výsledky vyhlazování a používá se jen výjimečně. Rozděluje příliš velké množství pravděpodobnostní masy. Pravděpodobnost pro neviděné $n-$gramy je nadhodnocená, zatímco pro viděné je podhodnocená. Všechny neviděné $n-$gramy mají stejnou pravděpodobnost.

Add lambda

Tato metoda vychází přímo z Add-One a částečně řeší její problém s rozdělením příliš velkého množství pravděpodobnosti. Místo jedna se k četnostem $n-$gramů přičte konstanta λ v rozsahu od nuly do jedné. Nedojde tak k neúměrnému zvýhodnění ne-

viděných $n-$gramů před viděnými. Konstanta se většinou získává z hold-out dat.

Vztah pro vyhlazení unigramu má podobu:

$$P_{+\lambda}(w) = \frac{C(w) + \lambda}{N + \lambda V}. \tag{17}$$

a obdobně pro bigram:

$$P_{+\lambda}(w_{n-1}w_n) = \frac{C(w_{n-1}w_n) + \lambda}{C(w_{n-1}) + \lambda V}. \tag{18}$$

Ač tato metoda odstraňuje nejpalčivější problémy svého předchůdce, stále mezi neviděné $n-$gramy rozděluje stejné množství pravděpodobnosti. Podávanými výsledky se nemůže vyrovnat sofistikovanějším metodám.

Witten-Bell

Mezi pokročilejší metody vyhlazování patří Witten-Bell. Metoda je založena na myšlence, že pravděpodobnost dosud neviděných $n-$gramů může být modelována pomocí pravděpodobnosti, že uvidíme $n-$gram poprvé. Díky tomuto předpokladu mohou být pravděpodobnosti $n-$gramů s nulovým výskytem odvozeny od pravděpodobností $n-$gramů s jedním výskytem, neboť oba jevy jsou si velmi blízké.

Celkovou pravděpodobnost unigramů s nulovým výskytem získáme ze vztahu:

$$\sum_{i:c=0} p_i^* = \frac{T}{(N+T)}. \tag{19}$$

N je celkový počet unigramů a T je počet nenulových různých unigramů.

Tuto pravděpodobnost rovnoměrně rozdělíme mezi nulové unigramy, kde Z představuje počet různých nulových unigramů:

$$Z = \sum_{i:c=0} 1 \tag{20}$$

$$p_i^* = \frac{T}{Z(N+T)} if(c_i = 0). \tag{21}$$

Masu pravděpodobnosti, kterou jsme rozdělili, musíme ubrat z nenulových unigramů, a proto přepočítáme pravděpodobnost nenulových unigramů pomocí následujícího vztahu:

$$p_i^* = \frac{c_i}{Z(N+T)} if(c_i > 0). \tag{22}$$

Pro bigramy platí podobné vztahy, ale změní se význam symbolů. $Z(w_x)$ je počet různých nulových bigramů začínajících slovem w_x, $T(w_x)$ představuje počet různých nenulových bigramů začínajících slovem w_x a $N(w_x)$ určuje počet bigramů, které začínají slovem w_x. Celkovou pravděpodobnost, kterou budeme rozdělovat, učíme za vztahu:

$$\sum_{i:c(w_x w_i)=0} p^*(w_i|w_x) = \frac{T(w_x)}{N(w_x)+T(w_x)}. \tag{23}$$

Tuto pravděpodobnost rozdělíme mezi nulové bigramy:

$$Z(w_x) = \sum_{i:c(w_x w_i)=0} 1, \tag{24}$$

$$p^*(w_i|w_{i-1}) = \frac{T(w_{i-1})}{Z(w_{i-1})(N+T(w_{i-1}))} if(c_i > 0). \tag{25}$$

A kvůli přerozdělení pravděpodobnosti přepočítáme a snížíme hodnoty nenulových bigramů:

$$\sum_{i:c(w_x w_i)>0} p^*(w_i|w_x) = \frac{c(w_x w_i)}{c(w_x)+T(w_x)}. \tag{26}$$

Pravděpodobnosti jsou rozdělovány na základě kontextu jednotlivých slov a nebudou pro všechny $n-$gramy stejné, jak tomu bylo u předchozích metod.

3.3.6 Kvalita jazykového modelu

Existuje velké množství postupů, jak vytvořit jazykový model, a podobná situace platí i pro metody vyhlazování. Abychom mohli porovnat různé jazykové modely, je potřeba mít k dispozici metodu pro měření kvalitu jazykového modelu.

Asi nejlepším způsobem by bylo různé modely zapojit do korektoru překlepů, provést sadu testů a porovnat výsledky úspěšnosti oprav. Takový přístup je však velmi zdlouhavý a často je výhodné měřit kvalitu jazykového modelu odděleně od ostatních částí korektoru.

Nejpoužívanější mírou pro ohodnocení kvality jazykového modelu je veličina zvaná perplexita. Tu můžeme přeložit do češtiny jako složitost a je definována:

$$PP = \frac{1}{\sqrt[K]{P(w_1 w_2 \dots w_K)}}. \tag{27}$$

Perplexitu si můžeme představit jako průměrný počet slov, mezi kterými se korektor rozhoduje v procesu opravy, používá-li daný jazykový model. Rovněž můžeme říci, že perplexita daného jazykového modelu odpovídá velkosti slovníku ekvivalentního jazykového uniformního modelu. Uniformní model je složený ze stejně pravděpodobných slov následujících za sebou v libovolném, ale stejně pravděpodobném pořadí.

Perplexita se často počítá pro trénovací i pro testovací korpus.

Mezi hlavní přednosti perplexity patří nezávislost na velikosti korpusu. Můžeme tak porovnávat modely s rozdílnou velikostí.

Pokud zlogaritmujeme perplexitu logaritmem se základem 2, získáme další důležitou veličinu – entropii, která vyjadřuje míru neuspořádanosti:

$$LP = \log_2 PP = -\frac{1}{K} \sum_{i=1}^{K} \log_2 P(w_i | w_1 w_2 \dots w_{i-1}). \tag{28}$$

Velká hodnota perplexity může poukazovat na nekvalitní model, ale také může spočívat ve velké neuspořádanosti (entropii) modelovaného jazyka. Jazyk s volnějšími pravidly bude mít obecně větší míru neuspořádanosti než jazyk s pevnými vazbami[3].

3.3.7 Jazykové modelování češtiny

Popsané metody tvorby jazykového modelu a vyhlazování jsou určeny především s ohledem na angličtinu. Problém řídkosti trénovacích dat je podstatně závažnější pro modelování českého jazyka.

Ohebnost

Čeština, podobně jako ostatní slovanské jazyky, patří mezi vysoce ohebné jazyky. To znamená, že pro každý základní tvar (lemma) existuje velké množství morfologických tvarů. Vlivem těchto změn může teoreticky vzniknout až 300 morfologických variant pro sloveso, 20 variant pro podstatné jméno a 200 variant pro přídavné jméno. Ve skutečnosti jsou tato čísla podstatně nižší. Přesto představuje ohebnost závažný problém, neboť každý tvar je v modelu uložen jako samostatné slovo. Velikost slovníku proto s narůstajícím objemem trénovacího korpusu roste a způsobí významné zvýšení počtu parametrů $n-$gramového modelu, který pak trpí nedostatkem dat pro natrénování.

Pokusy o vytvoření modelu založeného na morfologických principech se ukázaly být jako neefektivní. Za cenu značného zvýšení složitosti implementace nedošlo k výraznému zlepšení účinnosti na testovacích úlohách[2].

Volný pořádek slov ve větě

Další nepříjemnou vlastností pro jazykové modelování češtiny je volný pořádek slov ve větě. Po bližším prozkoumání se ukazuje, že jde spíše o slovosled variabilní, který se řídí sadou ustálených pravidel. Aby tato variabilita neměla negativní vliv na srozumitelnost sdělení, využívá čeština v hojné míře shodu mezi morfologickými kategoriemi jednotlivých větných členů. Například podmět se musí shodovat s přísudkem v osobě (muž psal), čísle a rodu, podstatné jméno a jeho přívlastek musí mít stejný pád, číslo a rod (mladé dívce). Nabízí se tedy využití morfologické shody při jazykovém modelování. Ke každému slovu můžeme připojit značku s popisem jednotlivých morfologických kategorií. Na základě těchto značek můžeme shlukovat slova do tříd a jazyk modelovat pomocí závislostí mezi třídami místo mezi slovy. Takto vytvořený model vykazuje vlastnosti typické pro modely založené na třídách – má menší počet parametrů a může být robustně natrénován, ovšem za cenu ztráty rozlišovací schopnosti. Při testování tohoto modelu na spisovné češtině bylo dosaženo slibných výsledků [2].

Nespisovný jazyk

Vážným problémem specifickým pro jazykové modelování češtiny je značný rozdíl mezi spisovnou a nespisovnou češtinou. Vzhledem k tomu, že v češtině dominuje fonetický princip, nejsou nespisovná slova pouhými výslovnostními variantami slov spisovných, ale tvoří samostatné morfologické tvary (*auťák*, *starej*), což nadále zvětšuje rozsah slovníku.

3.3.8 Jazykové modelování pro vyhledávač

Jazykový model použitý v úloze opravy dotazů zadávaných do vyhledávače se buduje z korpusu sestaveného z dotazů zadávaných do vyhledávače. S použitím takového typu dat se zajistí, že jazykový model bude věrněji odpovídat povaze řešené úlohy. Data získaná z dotazů jsou však velmi specifická a způsobují řadu problémů, které zapříčiní další zvýšení řídkosti trénovacích dat.

V mnoha úlohách se pro tvorbu jazykových modelů využívají tematicky zaměřené korpusy s omezeným slovníkem. Takový přístup není v řešené úloze možný, neboť

zadávané dotazy pokrývají široký záběr oborů a model musí být vybaven obsáhlým slovníkem.

Pro trénovací data je charakteristický i velký výskyt cizojazyčných výrazů, převážně anglických. V zadávaných dotazech se často objevují názvy obchodních značek a výrobků, jejich modelových a typových řad. Řada uživatelů vkládá do vyhledávače i www adresy.

Ve většině případech neprošla trénovací data žádnou jazykovou kontrolou a obsahují množství překlepů a chyb. Tyto chyby není možné technicky opravit nebo odstranit kvůli častému výskytu cizích slov. Každá chyba pak tvoří samostatný slovní tvar.

Problematickým a častým jevem je vynechávání diakritiky. Pokud v důsledku této chyby získáme nový slovní tvar (*řeřicha* na *rericha*), dojde k nežádoucímu zvýšení parametrů jazykového modelu. Vynechání diakritiky, jejímž zaviněním vznikne korektní známé slovo (*běž* na *bez*), způsobí zkreslení parametrů.

V řadě dotazů se setkáme i českou fonetickou transkripcí anglických slov, například: *google* na *gugl*, *notebook* na *noutbuk* nebo *cool* na *kul*. Tyto tvary opět zvyšují počet parametrů modelu.

Specifickým rysem dat získaných z dotazů je krátká historie slov, zastoupení slovních druhů a skladba. Uživatelský dotaz se v průměru skládá ze tří slov, což je méně než průměrná délka české věty. Mezi slovními druhy převažují podstatná jména, přídavná jména a příslovce. Výskyt sloves je výrazně nižší než v běžném jazyce. Mezi větným vztahy převažuje přívlastková a příslovečná vazba. Jak se uvedené skutečnosti projeví na kvalitě jazykového modelu, nebylo zjištěno.

Použitý trénovací korpus je synchronní a jeho složení odpovídá aktuálnímu společenskému, kulturnímu, sportovnímu, politickému a technickému dění. Pro zachování účinnosti je vhodné korpus pravidelně aktualizovat.

3.4 Chybový model

Chybový model se skládá z pravidel a vztahů popisujících, jak se ze správného slova stane chyba.

Pomocí chybového modelu se pokusíme z chybně napsaného tvaru odvodit seznam kandidátů slova, které chtěl uživatel původně napsat. Seznam možných kandidátů může být velmi rozsáhlý, a proto je musíme umět ohodnotit. Protože se v řešené úloze držíme především stochastických principů, budeme kandidáty ohodnocovat především pravděpodobností.

3.4.1 Klasifikace chyb

Chyby můžeme rozdělit pomocí různých hledisek do několika kategorií. Různé typy chyb vyžadují specifické metody k jejich detekci a korekci.

Literatura zabývající se opravou překlepů nejčastěji rozděluje chyby podle pisatelova zavinění na kognitivní chyby a překlepy. U překlepů uživatel zná správný tvar slova, ale například přehmatem na klávesnici jej napsal chybně (*p5eklep*). Kognitivní chyby jsou způsobeny neznalostí správného tvaru slova (*bilina*) a jedná se především o gramatické chyby.

Speciálním druhem kognitivních chyb jsou chyby homofonní, kdy pisatel zaměňuje znaky, nebo dokonce celá slova za stejně znějící. Tento druh chyb je charakteristický zejména pro angličtinu a francouzštinu[4]. Příklady některých anglických záměn jsou: *base* a *bass*, *two* a *too*, *break* a *brake*, *piece* a *peace*. Existuje několik algoritmů (SOUNDEX, METAPHONE), které se speciálně zaměřují na fonetické chyby a dosahují vysoké úspěšnosti. Rozšířený nástroj Aspell s úspěchem využívá algoritmu METAPHONE pro korekci anglických textů. V českém jazyce je frekvence výskytu tohoto druhu chyb zanedbatelná a převážně se jedná o záměnu *s* a *z*.

Znalost, zda je chyba kognitivní nebo překlep, by nám usnadnila korekci. U kognitivních chyb bychom se zaměřili na gramatické vztahy a u překlepů můžeme vycházet z rozložení klávesnice. Bohužel ve většině případech nejsme schopni určit, o jaký druh chyby se jedná, proto v rámci diplomové práce provedeme zjednodušení a oba typy chyb budeme považovat za překlepy.

Velmi problematická je kategorie chyb způsobující rozdělení (*roz vod*) nebo spojení slov (*vpřírodě*). K jejich korekci je potřeba použít sofistikované metody. Statistiky ukazují, že tento druh chyb není příliš častý, a proto je řada korektorů ignoruje[5].

Mezi nejzáludnější patří chyby, jejichž výsledkem je opět správné slovo. V angličtině se tento druh označuje termínem *real-word errors*. Takovou chybu získáme například, pokud ve slově píseň zaměníme *ň* za *k*, a získáme tak *písek*. S tímto druhem chyb se umí korektor účinně vypořádat pouze v případě, že využije kontext okolních slov. Pak je zřejmé, že hudebník složil *píseň*.

3.4.2 Přístup na základě editační vzdálenosti

Abychom uměli popsat vzájemnou odlišnost dvou různých řetězců, musíme zavést vhodnou metriku. Metriky používané v korektorech překlepů bývají často založené na minimální editační vzdálenosti. Editační vzdálenost se definuje pomocí sady ope-

rací, které umožňují převedení jednoho řetězce na druhý. Takto definovaná metrika je vhodná pro popis překlepů, neboť jednotlivé operace často korespondují s různými druhy překlepů. Nejpoužívanější metrikou je Damerau-levenshteinova vzdálenost.

Damerau-levenshteinova vzdálenost. *Jedná se o minimální počet editačních operací, které musíme vykonat, abychom převedli řetězec w na c, kde editační operace jsou záměna (přepsání) znaků, smazání znaku, vložení znaku a transpozice dvou sousedních znaků.*

Tabulka 1. Příklady editačních operací

Editační operace	chyba
záměna	ch**i**ba
smazání	ch_ba
vložení	chy**r**ba
transpozice	ch**by**a

V literatuře se často setkáme i s pojmem Levenshteinova vzdálenost. Ta se od Damerau-levenshteinovy vzdálenosti liší tím, že mezi editační operace nezařazuje transpozici znaků.

Algorimus pro výpočet minimální editační vzdálenosti si popíšeme a předvedeme pro Levenshteinovu vzdálenost, neboť je názornější a jednodušší. S drobnou úpravou lze algoritmus využít i pro výpočet Damerau-levenshteinovy vzdálenosti.

Algoritmus pro výpočet Levenshteinovy vzdálenosti

Pro výpočet Levenshteinovy vzdálenosti se používá algoritmus založený na principu dynamického programování. Dynamické programování je označení pro třídu tabulkově řízených algoritmů, které řeší rozsáhlé problémy vhodnou kombinací řešení podproblémů, jejichž výsledky jsou uloženy v tabulce.

Algoritmus pro výpočet Levenshteinovy vzdálenosti nejdříve vytvoří matici, ve které počet řádků odpovídá délce zdrojového řetězce a počet sloupců délce cílového řetězce. Každý prvek matice na pozici i a j bude obsahovat vzdálenost mezi prvními i znaky zdrojového a prvními j znaky cílového řetězce. Výpočet prvků matice začíná v levném horním rohu a končí v pravém dolním rohu, ve kterém je uložena výsledná minimální vzdálenost. Hodnota jednotlivých prvků se vždy určí ze tří okolních jednoduchou funkcí, kdy vybíráme nejmenší ze tří možných cest při průchodu maticí:

$$matice[i,j] = min \begin{cases} matice[i-1,j] + cena\_mazani \\ matice[i-1,j-1] + cena\_zameny \\ matice[i,j-1] + cena\_vlozeni \end{cases} \tag{29}$$

Algoritmus zapsaný v pseudokódu vypadá následovně:

```
function levenshtein_distance(target, source) returns min_distance
  n= length(target)
  m = length(source)
  create a distance matrix distance[n+1, m+1]
  distance[0,0] = 0
  for each column i from 0 to n do
    for each column j from 0 to m do
      distance[i, j] = min(distance[i-1,j] + DEL_COST, //deletion
                           distance[i-1,j-1] + SUB_COST, //subtition
                           distance[i,j-1] + INS_COST)) //insertion
```

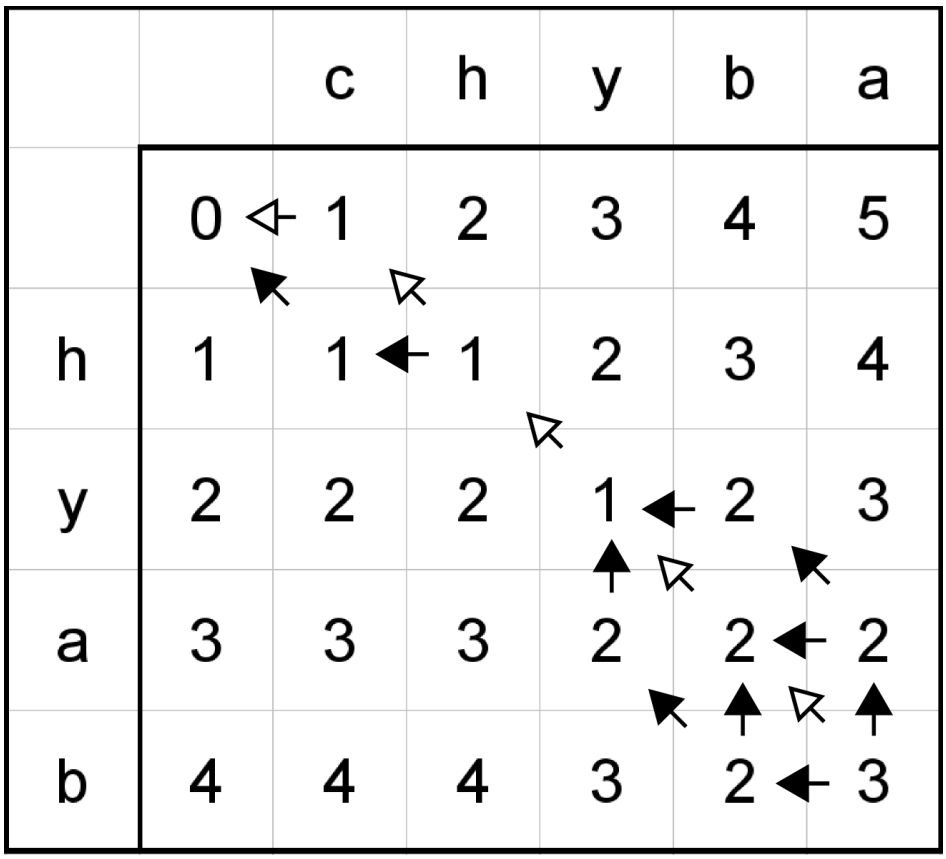
Na obrázku 1. je uveden příklad výpočtu vzdálenosti slov *hyab* a *chyba*. Význam šipek bude osvětlen záhy.

Často potřebujeme nejen vědět jaká je mezi řetězci vzdálenost, ale i jak se vzájemně liší. K popisu rozdílů mezi řetězci se používají nejčastěji dva způsoby: zarovnání a seznam operací.

Zarovnání – Při zarovnání upravíme řetězce tak, aby měly stejnou délku. Při operaci smazání vložíme prázdný znak do cílového slova (*tunel* a *t_nel*), při vkládání do slova zdrojového (*tu_nel tuunel*).

Seznam operací – Posloupnost editačních operací potřebných k tomu, abychom převedli zdrojové slovo na cílové.

Oba popisy rozdílů mezi řetězci snadno získáme z matice pro výpočet minimální editační vzdálenosti. Začneme v pravém dolním rohu a vždy budeme vybírat nejmenší ze tří možných cest, dokud se nedostaneme do levého horního rohu. Krok vlevo znamená přidání znaku, při kroku vpravo znak mažeme a cestou po diagonále znak měníme. Musíme dát pozor na to, že zarovnání a seznam operací získáme v opačném pořadí a musíme je otočit.



Obrázek 1. Příklad výpočtu levenstainovy vzdálenosti

V původním příkladu jsou nakresleny šipky, které zobrazují možné cesty. Jak je vidět, cesta nemusí být vždy jednoznačná, a můžeme tak získat několik různých zarovnání a seznamů operací.

Jestliže se v uvedeném příkladu vydáme po nevyplněné šipce, získáme následující zarovnání:

_**hyab** a **chyba**

a seznam operací:

- **vložení** *c*
- **záměna** *a* za *b*
- **záměna** *b* za *a*.

Nyní se vrátíme k Damerau-levenshteinovy vzdálenosti. Jak už bylo napsáno výše, liší se od Levenshteinovy vzdálenosti tím, že navíc obsahuje operaci transpozice, kterou jednoduše doplníme.

```
function damerau_levenshtein_dist(target, source) returns min_dist
  n = length(target)
  m = length(source)
  create a distance matrix distance[n+1, m+1]
  distance[0,0] = 0
  for each column i from 0 to n do
    for each column j from 0 to m do
      distance[i, j] = min(distance[i-1,j] + DEL_COST, //deletion
                           distance[i-1,j-1] + SUB_COST, //subtition
                           distance[i,j-1] + INS_COST)) //insertion
      if(i > 1 and j > 1 and source[i] = targer[j-1]
        and source[i-1] = target[j]) then
        distance[i,j] = min(distance[i, j],
                            distance[i-2,j-2]+TRN_COST)//transposition
```

Výše uvedený příklad by měl Damerau-levenshteinovu vzdálenost 2 a seznam operací by byl:

- **vložení** *c*
- **transpozice** *b* za *a*.

Cena operací

Většinou předpokládáme, že cena všech operací je stejná, tedy 1. V některé literatuře se doporučuje zvýšit cenu záměny znaků na 2, neboť jedna záměna nahradí dvojici operací přidání a smazání[4]. Často se využívá i komplexnějšího ohodnocení operací, například pomocí pravděpodobností. Při ohodnocení pravděpodobností už nebudeme o mluvit o minimální editační vzdálenosti, ale o *nejpravděpodobnějším zarovnání řetězců*. Nejpravděpodobnějšího zarovnání využíváme i v našem korektoru.

Získání kandidátů

Jedním z úkolu chybového modelu je odvození seznamu možných kandidátů na opravu z chybného slova. Při metodě založené na minimální editační vzdálenosti samozřejmě využijeme editační operace.

Seznam kandidátů získáme k-násobnou aplikací všech možných variant editačních operací na opravované slovo, kde k je zvolená editační vzdálenost. Takto vygenerovaný

seznam obsahuje všechny možné slovní tvary do editační vzdálenosti k, ze kterých mohlo opravované slovo vzniknout. To znamená, že například pro $k = 1$ při operaci vkládání vložíme postupně na všechny pozice opravovaného slova všechny znaky použité abecedy. Při mazání odebereme z každé pozice jeden znak slova.

Výsledný seznam je velmi rozsáhlý a jen pro vzdálenost 1 existuje $(2r + 2)n + r$ možných tvarů, kde n je délka slova a r počet znaků použité abecedy. Pokud bychom uvažovali pouze anglickou abecedu, bude se jednat o $54n+26$ slovních tvarů. V případě české abecedy počet stoupne dokonce na $88n + 43$.

Ohodnocení kandidátů

Ohodnocování kandidátů pomocí chybového modelu není nic jiného než výpočet nejpravděpodobnějšího zarovnání kandidáta a původního slova. Při ohodnocování kandidátů budeme vycházet z parametrů, podle kterých byl kandidát vygenerován. Kandidáta lze ohodnotit různou úrovní přesnosti. Při nejjednodušším ohodnocení vezmeme v potaz pouze editační vzdálenost.

Pro zvýšení přesnosti můžeme uvažovat i typy editačních operací použitých při generování kandidáta, například jestli byl kandidát získán smazáním, nebo vložením znaku.

Pokud bychom chtěli dosáhnout dalšího zvýšení přesnosti, zahrneme do ohodnocení i znaky, na které byly editační operace aplikovány, případně i jejich pozici ve slově. V tomto případě využijeme znalosti, kdy víme, že záměna znaků i a y je mnohem častější než například záměna r za t nebo r za y.

Vzorec pro výpočet ohodnocení kandidáta dle chybového modelu vypadá následovně:

$$P(w|c) = \prod_{i=1}^{l} \prod_{j=1}^{k} P(e_{ij}), \tag{30}$$

kde k je editační vzdálenost a l úroveň přesnosti.

Některé pokročilejší metody s úspěchem využívají k ohodnocení i kontext okolních znaků a mohou využít znalosti, kdy například vědí, že y je častěji smazané po b než po a.

Trénovaní chybového modelu

Při trénovaní chybového modelu nejčastěji využíváme seznam dvojic, které se skládají vždy ze správného tvaru a překlepu. Na základě tohoto seznamu pak uděláme statistickou analýzu chyb, kterou použijeme na tvorbu chybového modelu.

3.4.3 Přístup na základě fonetické podobnosti

Přestože metody založené na fonetické podobnosti nejsou příliš účinné pro český jazyk, s úspěchem se používají pro opravy anglických textů. Tyto metody jsou často zmiňovány v literatuře věnující se opravě překlepů a setkáme se s nimi i v řade dostupných korektorů, například v Aspellu. Proto si o dvou z nich řekneme několik slov a krátce si je popíšeme.

Soundex

Tento algoritmus vznikl již v roce 1922 a sloužil pro indexaci jmen v kartotékách pro telefonní operátory. Jména se indexovala podle výslovnosti, a operátor tak mohl vyhledat záznam volajícího rychleji, než kdyby se jméno hláskovalo.

Algoritmus převede každé slovo na čtyřznakový kód, který začíná písmenem následovaným třemi číslicemi. Kód se získá tak, že se první písmeno znaku ponechá a ostatní se převedenou podle tabulky 2 na čísla. Znaky, které nejsou v tabulce, se vynechají, stejně jako po sobě se opakující číslice, vyjma první. Z takto získaného tvaru se ponechají pouze první čtyři znaky. V případě, že je kód příliš krátký, doplní se nulami. Například pro jméno Robert bude kód R163.

Tabulka 2. Pravidla převodu Soundex

Znak	Kód
A, E, I, O, U, H, W, Y	-
B, F, P, V	1
C, G, J, K, Q, S, X, Z	2
D,T	3
L	4
M, N	5
R	6

V případě, že bychom chtěli využít Soundex pro modelování chyb, pak seznam kandidátů na opravu sestavíme ze slov se stejným kódem. Soundex neumožňuje

kandidáty ohodnotit.

Přestože je SOUNDEX zastaralý a pro opravu chyb téměř nepoužitelný, byl první a stal se vzorem pro moderní algoritmy založenými na fonetické podobnosti.

Metaphone

METAPHONE vychází z algoritmu SOUNDEX a opravuje řadu jeho nedostatků. Poskytuje přesnější fonetické kódovaní slov, neboť využívá propracovanější a rozsáhlejší sadu pravidel pro přepis výslovnosti. Základem algoritmu je šestnáctiznaková abeceda: B X S K J T F H L M N P R 0 W Y. Pravidla nebudeme kvůli jejich rozsahu uvádět, ale dají se najít například na [6]. Výsledný kód slova nemá na rozdíl od SOUNDEX pevně stanovenou délku.

V chybovém modelu bychom využili METAPHONE podobně jako SOUNDEX, kde kandidáty získáme jako slova se stejným kódem. METAPHONE je velmi výkonný algoritmus a s úspěchem je využíván v programu Aspell.

3.4.4 Další přístupy

V literatuře věnující se korektuře překlepů často narazíme na řadu dalších metod. Většinou se však jedná o obměnu nebo kombinaci dvou výše uvedených principů. Najdou se však i metody výrazně odlišné od těch, se kterými jsme se doposud seznámili.

n-gramový přístup

V chybovém modelování budeme pod termínem $n-$gram rozumět posloupnost znaků. Například slovo *fakta* bude rozděleno na následující bigramy: *-f fa ak kt ta a-* a trigramy: *-fa fak akt kta ta-*. $n-$gramový přístup zahrnuje celou škálu různých metod jak deterministických, tak stochastických.

Pokud bychom chtěli použít $n-$gramový přístup ke získání kandidátů, museli bychom například z chybové analýzy zjistit jaké $n-$gramy se často zaměňují, nebo bychom mohli využít editační vzdálenosti, na opravované slovo aplikovat editační operace a následně jej rozložit na $n-$gramy.

Ohodnocení je možné udělat podle znakového $n-$gramového modelu nebo určit podobnost kandidáta s původním slovem. Podle nejjednoduššího vztahu se podobnost dvou $n-$gramů spočítá jako počet shodných $n-$gramů dělených celkovým počtem $n-$gramů.

$n-$gramové metody nejsou příliš účinné na krátká slova a nefungují na asijské jazyky.

Umělá inteligence

Většina úvah, ohledně využití umělé inteligence pro opravu překlepů, stále zůstává v teoretických rovinách. Funkční prototypy, přestože dosahují dobrých výsledků, jsou kvůli problémům s trénováním omezené na specifická témata a obecně nepoužitelné[4].

Dílčích úspěchů se povedlo dosáhnout v oblasti strojového učení použitím algoritmu Winnow. Úspěšnost tohoto korektoru překonává algoritmy založené na statistických metodách i pro obecné texty. Bližší popis algoritmu je mimo rozsah práce. Podrobnosti lze nalézt v [7].

3.5 Algoritmus korekce

Nyní už jsme se seznámili se všemi pilíři korektoru a můžeme přejít k vlastnímu algoritmu.

Zadaný uživatelský dotaz se nejdříve vyčistí a rozdělí na slova. Následně se jednotlivá slova postupně procházejí a pomocí korekčního algoritmu se snažíme najít jejich správný tvar. Jakmile opravíme všechna slova, složíme z nich zpět opravený dotaz.

Algoritmus korekce slova probíhá ve čtyřech fázích:

1. vytvoření seznamu kandidátů
2. ohodnocení kandidátů dle chybového modelu
3. ohodnocení kandidátů dle jazykového modelu
4. výběr nejvhodnějšího kandidáta

3.5.1 Zpracování dotazu

Dotaz se rozdělí podle bílých znaků, čárek, teček a podtržítek na jednotlivá slova, která se následně převedou na malá písmena. Slova obsahující více než 50% číslovek se stejně jako jednopísmenné tvary přeskakují.

3.5.2 Vytvoření seznamu kandidátů

Použitý chybový model je založen na editační vzdálenosti a kandidáty tedy získáme pomocí editačních operací. Z výkonnostních důvodů budeme generovat kandidáty pouze do vzdálenosti 2 a ponecháme si jen ty, které známe z unigramového modelu.

3.5.3 Ohodnocení kandidátů dle chybového modelu

Kandidáty budeme ohodnocovat na čtyřech úrovních přesnosti. Vyšší úroveň by měla zajistit přesnější ohodnocení, ale je náročnější na výpočetní výkon.

Nultá úroveň představuje naivní přístup. Pokud se opravované slovo nachází v unigramovém modelu, je považováno za jediného kandidáta. Jestliže je opravované slovo neznámé, algoritmus se podívá, zda existují kandidáti do vzdálenosti 1. V kladném případě postoupí tyto kandidáty do další fáze algoritmu a stávající fáze je ukončena. V opačném případě se podívá po kandidátech do vzdálenosti 2. Pokud je i tento seznam prázdný, je oprava ukončena. Předností této úrovně je rychlost a jednoduchost, ale je zřejmé, že nebere v potaz získané chybové statistiky. Jak uvidíme později, její úspěšnost není příliš vysoká.

Na první úrovni přesnosti se při ohodnocování kandidátů bere v potaz pouze jejich editační vzdálenost.

Druhá úroveň už do ohodnocování kandidátů započítává i použitou editační operaci. Pokud je vzdálenost kandidáta větší než 1, je tato hodnota z důvodu výpočetní náročnosti aproximována průměrnou cenou operací.

Třetí úroveň bere v úvahu i parametr editační operace, tedy konkrétní znak. Stejně jako u předchozí úrovně je tato hodnota u kandidátů se vzdáleností 2 a více aproximována.

3.5.4 Ohodnocení kandidátů dle jazykového modelu

Z jazykového modelu určíme apriorní pravděpodobnost kandidáta. Budeme rozlišovat, zda je opravované slovo izolované, nebo se nachází v kontextu dalších slov. Pokud je slovo izolované, přiřadíme kandidátovi pravděpodobnost z unigramového modelu.

V případě, že se opravované slovo nachází v kontextu dalšího slova, určíme podmíněnou pravděpodobnost dvojice z bigramového modelu. V tomto výpočtu na nás čeká drobná záludnost. Většinou předpokládáme, že opravované slovo následuje po kontextovém slově. První slovo dotazu však nemá žádný předcházející kontext, ale má

následující. Pro první slovo dotazu budeme tedy počítat podmíněnou pravděpodobnost, že opravované slovo bude následováno kontextovým.

3.5.5 Výběr nejvhodnějšího kandidáta

V posledním kroku jednoduše vybereme kandidáta, jehož součin ohodnocení je dle jazykového a chybového modelu největší.

3.5.6 Příklad

Abychom získali lepší představu o principu, na jakém algoritmus funguje, předvedeme si jednoduchý příklad.

Uživatel do vyhledávače vložil dotaz *Kertl Gott*. Je zřejmé, že uživatel chtěl původně napsat *Karel Gott*, ale omylem se dopustil dvou překlepů.

Nejprve dotaz rozdělíme na slova, která následně převedeme na malá písmena, a získáme: *kertl* a *gott*.

Nyní už můžeme použít opravný algoritmus na slovo *kertl*.

1. Nejdříve vygenerujeme seznam možných kandidátů na opravu.

   Pro editační vzdálenost 1 jsme získali následující kandidáty (slova jsou ze skutečného jazykového modelu):

   kert (smazání l), kartl (záměna e za a), mertl (zaměna k za m) ...

   Počet kandidátů pro editační vzdálenost bude podstatně vyšší, ale vybereme jen prvních pár:

   kuril (záměna e za u; záměna t za i), certe (záměna k za c; záměna t za i), karel (záměna e za a; záměna t za e) ...

2. Kandidáty ohodnotíme dle chybového modelu.

   Ze statistiky víme, že kandidáti do vzdálenosti 1 jsou pravděpodobnější než do 2, že nahrazení znaků je pravděpodobnější než transpozice atd.

   kert $(0,0096)$, kartl $(0,0224)$, mertl $(0,00167)$

   kuril $(0,0000256)$, certe $(0,0000743)$, karel $(0,000315)$

3. Kandidáty ohodnotíme dle jazykového modelu s využitím kontextu slova gott

   kert $(1,84.10^{-6})$, kartl $(5,78.10^{-5})$, mertl $(1,84.10^{-6})$

   kuril $(3,98.10^{-4})$, certe $(2,8.10^{-4})$, karel $(3,57.10^{-3})$

4. Pravděpodobnosti vynásobíme a získáme celkové ohodnocení.

   kert $(1,76.10^{-8})$, kartl $(1,30.10^{-6})$, mertl $(3,07.10^{-9})$

   kuril $(1,01.10^{-8})$, certe $(2,08.10^{-8})$, karel $(1,12.10^{-6})$

   Vidíme, že největší ohodnocení má slovo karel.

Dále bychom provedli korekci pro slovo *gott*, které by jistě bylo opraveno opět na *gott*.

Na závěr vrátíme opravený dotaz *karel gott*. Ve skutečném provozu bychom museli slovům vrátit správnou velikost písmen, zde si však dovolíme drobný ústupek kvůli jednoduchosti.

3.6 Další algoritmy

Krátce si popíšeme několik nejznámějších korektorů a seznámíme se s jejich korekčními algoritmy a strategiemi.

3.6.1 Ispell

Ispell je jedním z nejstarších korektorů a jeho historie se dá vystopovat až do sedmdesátých let dvacátého století. Jeho algoritmus je založena na Damerau-levenshteinovy vzdálenosti, generuje kandidáty pouze do vzdálenosti 1 a všechny ohodnocuje stejně. Takto jednoduchý algoritmus není příliš účinný, a proto se v současné době Ispell téměř nepoužívá.

3.6.2 GNU Aspell

GNU Aspell, často také zkráceně nazývaný pouze Aspell, je svobodný korektor překlepů vytvořený jako náhrada programu Ispell. Jeho korekční strategie je založena na kombinaci přístupu fonetické podobnosti a Damerau-levenshteinovy vzdálenosti. Pro korekci anglických textů se využívá algoritmus Metaphone. V současnosti se jedná o nejrozšířenější korektor ve světě svobodného software.

1. Korekce probíhá podle následujícího postupu:

2. Opravované slovo se převede na fonetický kód.

3. Vygenerujeme seznam všech fonetických kódů do editační vzdálenosti 2 od původního kódu a ponecháme si pouze ty, které známe ze slovníku kódu. Všechna slova, jejichž fonetický kód se nachází ve výše popsaném seznamu, přidáme do seznamu kandidátů na opravu.

4. Každé editační operaci je přiřazena cena. Pro jednotlivé kandidáty určíme oceněnou editační vzdálenost a spočítáme stejnou vzdálenost i pro jejich zvukové kódy. Na závěr ohodnotíme kandidáty váženým průměrem dvou získaných oceněných editačních vzdáleností.

5. Vybereme slova s nejnižším ohodnocením.

3.6.3 Myspell

Tento korektor byl původně určen pro kancelářský balík OpenOffice. Podobně jako řada ostatních korektorů vychází z Ispell, který dále rozšiřuje o možnost komprese morfologických přípon.

3.6.4 Hunspell

Hunspell je korektor překlepů a zároveň morfologický analyzátor zaměřený na jazyky s bohatým tvaroslovím. Původně byl určen pro maďarštinu, odkud pochází i jeho název.

Hunspell vychází z Myspell a je zpětně kompatibilní s jeho slovníky. V současnosti je tento korektor výchozí pro programy OpenOffice, Firefox 3 a Thunderbird.

4 Implementace

Jedním z požadavků zadání bylo vytvoření funkčního prototypu korektoru. Právě tato kapitola se mu bude z velké části věnovat. Seznámíme se s jeho implementací a podobně si popíšeme jednotlivé moduly, třídy, funkce a skripty, které využívá. Tento popis bude sloužit zároveň i jako dokumentace a manuál k programu.

Pro tvorbu korektoru byl zvolen programovací jazyk Python verze 2.5, který je se svými vlastnostmi předurčen pro tvorbu prototypů. Podrobněji se o něm rozepíšeme v úvodní části kapitoly.

Rovněž se budeme věnovat tvorbě jazykového a chybového modelu. Celý proces tvorby bude podrobně popsán včetně použitých skriptů tak, aby jej bylo možné opakovat.

Všechny zdrojové kódy jsou součástí CD přílohy diplomové práce.

4.1 Python

Python je dílem holandského programátora Guido van Rossum. Pan Rossum poprvé Python představil v roce 1989 a od té doby se zněj stal vyspělý, stabilní a zralý jazyk vhodný pro nasazení v komerční i akademické sféře. Od svého zrodu získal desetitisíce přívržen̊ců a bylo v něm vytvořeno bezpočet aplikací.

Python disponuje mnoha vlastnostmi moderních programovacích jazyků – je intepretovaný, objektově orientovaný, vybaven automatickou správou paměti a součástí základní distribuce je bohatá standardní knihovna. Jednoduchá syntax jazyka se snadno učí a programy v něm napsané jsou přehledné a čitelné. Díky těmto vlastnostem získal Python pověst nejproduktivnějšího jazyka na světě. O zralosti Pythonu svědčí jeho využití v mnoha velkých institucích, mezi které patří NASA, Google, Yahoo nebo Red Hat.

4.1.1 Základní rysy jazyka

Několik významných vlastností jazyka bylo zmíněno výše, nyní si je popíšeme podrobněji a zmíníme několik dalších.

Kombinace paradigmat

Python dosahuje vysoké produktivity a efektivity práce, především unikátní kombinací principů různých programovacích paradigmat. V Pythonu nalezneme prvky znásledujících paradigmat: imperativní, objektově orientované a funkcionální.

Výchozí způsob programování v Pythonu je imperativní. Od imperativního programovaní můžeme snadno přejít k objektově orientovanému, protože jazyk je od základu objektový. O objektovém paradigmatu se rozepíšeme dále v textu.

Řadu imperativně komplikovaných úloh lze snadno řešit za pomoci funkcionálních principů, které zvyšují expresivitu a usnadňují programátorovi práci.

Většina funkcionálních rysů jazyka se využívá především při manipulaci s datovým typem seznam. K tomuto účelu je Python vybaven aparátem, kterému se říká stručný seznam. Stručným seznam umožňuje doslova čarovat se seznamy a v řadě programovacích konstrukcí nám ušetří mnoho řádků kódu.

Přenositelný

Za významné rozšíření vděčí jazyk především své dostupnosti na nejrozmanitějších platformách. Standardní implementace Pythonu je naprogramovaná v čistém ANSI C a lze ji zkompilovat a spustit na všech dnešních významných platformách. S Pythonem se můžeme setkat na UNIX, Linux, MS-DOS, MS Windows (95, 98, NT, 2000, XP), Macintosh, Amiga, AtariST, Be-OS, OS/2, VMS, PalmOS, PocketPC a mnoha dalších.

Standardní knihovny jsou napsané tak, aby se programátor nemusel zabývat detaily operačního systému nebo platformy. Programy vytvořené v Pythonu se překládají do přenositelného bajtkódu a lze je spustit na libovolné platformě s kompatibilní verzí interpretu.

Z toho vyplývá, že pokud budeme při tvorbě programu využívat pouze standardní knihovny, spustíme program na MS Windows i na Linux bez změny jediné řádky.

Objektově orientovaný

Od verze 2.2 byly základy jazyka přepracovány na čistě objektový systém. Číslo, řetězec, seznam, datový typ jsou všechno objekty stejně jako třída. Objektový model v Pythonu podporuje následující principy: zapouzdření, dědičnost, polymorfismus, přetěžování operátorů a metatřídy.

Čitelný

V jazyce Python se určuje struktura bloků pomocí odsazování, což se pozitivně projeví na přehlednosti a čitelnosti zdrojového kódu. Programy od různých lidí mají podobnou strukturu, a i začínající programátoři se vyznají ve zdrojových kódech zkušených kolegů. Díky dobré čitelnosti jazyka se programátoři dopouštějí měně chyb.

Bohatá standardní knihovna

V souvislosti s filozofií standardních knihoven získal Python přívlastek: "Batteries included", přeloženo jako: "baterie součástí balení", neboť součástí instalace Pythonu je i bohatá výbava knihoven, která pokrývá široký rozsah programátorských problémů.

Součástí standardní knihovny jsou moduly pro práci s regulárními výrazy, XML, databázemi, vlákny, síťovými protokoly nebo multimédii. Pro tvorbu aplikací s grafickým uživatelským rozhraním, je k dispozici modul Tkinker, což je rozhraní Pythonu pro knihovnu Tk.

Samozřejmě nejsme odkázáni jen na standardní knihovnu. Existují stovky knihoven výrobců třetích stran a modulární architektura jazyka umožňuje jejich snadné včlenění do našich aplikací.

Svobodný

K širokému rozšíření jazyka pomohlo i to, že je svobodný, otevřený, a tedy i zdarma. Zdrojový kód Pythonu můžeme libovolně kopírovat, měnit, nebo dokonce i prodávat. Licence jazyka je kompatibilní s GPL.

Svobodný v případě Pythonu neznamená nepodporovaný. Opak je pravdou. Díky rozšířenosti jazyka se odpovědi na dotaz často dočkáme stejně rychle jako od komerčních dodavatelů software.

4.1.2 Technické rysy jazyka

Svou kompilační strategií se Python podobně jako Java řadí mezi hybridní jazyky. Zdrojový program se kompiluje do mezikódu (bytecode), který je předán virtuálnímu stroji a následně interpretován.

Python mistrně kombinuje přednosti skriptovacích a kompilovaných jazyků. Poskytuje snadnost používání skriptovacích jazyků, ale na rozdíl od nich obsahuje pokročilé nástroje kompilovaných jazyků a díky tomu je možné vytvářet i velké projekty.

Automatická správa paměti

Python obstarává všechny nízkoúrovňové detaily okolo správy paměti. Jakmile se nějaký objekt stane nepotřebný, Python ho automaticky odstraní a uvolní po něm paměť.

Dynamické typování

Během programování v jazyce Python nemusíme během deklarace uvádět typ a ani velikost proměnné, všechno určí virtuální stroj za běhu. Pokud za běhu dojde například ke změnám rozměru seznamu, Python sám automaticky alokuje nový prostor, překopíruje do něj jednotlivé položky a předchozí paměťový prostor uvolní.

Rapidní vývoj aplikací

Řada zmíněných vlastností činní Python výborným kandidátem pro RAD. Hybridní nátura Pythonu nám umožní přeskočit zdlouhavé fáze kompilovaní a významně tak zrychlí vývoj. To nám například umožní získat okamžitou odezvu po provedení změn v aplikaci.

Krátký životní cyklus programu není jediný důvod, proč je vývoj v jazyce Python tak rychlý. Svou zásluhu má i jednoduchá syntax, vestavěné nástroje a další vlastnosti popsané v okolním textu. Často se Pythonu přezdívá: "spustitelný pseudokód". Programátor se může plně věnovat řešení problému místo toho, aby zápasil se záludnostmi programovacího jazyka. Rovněž je ověřeno, že program napsaný v Pythonu bude podstatně kratší než funkčně ekvivalentní program v jazyce C nebo Java.

Široké aplikační uplatnění

Jen málokterý jiný jazyk se může pyšnit takovou rozmanitostí aplikací. Python je možné použít pro psaní krátkých skriptů, ale i vývoj kolosálních aplikací o sto tisíců řádek.

V Pythonu můžeme psát administrativní skripty pro správu systému. Oborem, kde lze Python úspěšně využít, jsou numerické vědecké a inženýrské výpočty. Pokud by nám pro tvorbu grafických uživatelských nevyhovoval Tkinker, máme na výběr zmnoha dalších knihoven. Jsou připraveny rozšíření pro GTK+, QT, WXWindows nebo XUL.

V poslední době se pozornost softwarového průmyslu soustřeďuje na internetové aplikace. I v tomto odvětví Python exceluje a nabízí širokou škálu přístupů. Můžeme

programovat v CGI skriptech, vyvíjet v mamutím aplikačním serveru ZOPE nebo využít některý zmnoha rapidních webových frameworků.

4.2 Programové vybavení

Programové vybavení obsahuje balíček jazyka Python a sadu skriptů.

4.3 Balíček spell-correction

Balíček obsahuje sadu modulů pro stochastické jazykové modelování. Je tedy možné jej využít i pro jiné úlohy statistické lingvistiky, než je oprava překlepů.

Balíček spell-correction byl vytvořen pomocí standardních distribučních nástrojů jazyka Python za použití modulu distutils. Před vlastním používáním korektoru a pomocných skriptů musí být nainstalován do systému. Instalace se provádí pomocí následující posloupnosti příkazů:

```
gunzip -c spell-correction-1.0.tar.gz|tar xf -

cd spell-correction-1.0

python setup.py install
```

O dalších možnostech instalace je více uvedeno na [8].

Následuje popis jednotlivých modulů včetně nejdůležitějších tříd a funkcí.

4.3.1 language_model

Modul je určen pro práci s jazykovým modelem. Obsahuje jedinou třídu LanguageModel

LanguageModel

Třída obstarává všechny náležitosti spojené s jazykovým modelem jako je vyhlazování, počítání relativních četností, výpočet entropie atd.

Základem modelu je unigramový model (unigrams), bigramový model (bigrams) a inverzní bigramový model (ibigrams). Pro uložení všech modelů byla použita datová struktura (dict) neboli slovník.

V unigramovém modelu jsou unigramy uloženy jako klíče slovníku a jejich četnost v korpusu jako hodnoty klíčů.

V bigramovém modelu musíme umět uložit dvojici po sobě jdoucích slov. K tomu využíváme vnořené slovníky. Klíče ve slovníku (bigrams) odkazují na další slovníky obsahující možné následníky a jejich četnost.

V inverzním bigramovém modelu máme uloženy četnosti předcházejících dvojic slov. Tuto informaci bychom samozřejmě uměli získat i z bigramového modelu, ale to bylo by výpočetně velmi náročné.

Slova v modelu mohou obsahovat pouze běžné znaky abecedy a &/:?.-.

`__init__(self, error_model)`

Parametrem konstruktoru je chybový model, který se používá k ohodnocení kandidátů na opravu. Více se o korekci rozepíšeme v modulu correction.

`load_from_timestamp_file(self, path, encoding="utf-8", min_len=1)`

Naplní bigramový a unigramový model z textového korpusu s cestou (path). Jako korpus je použit záznam uživatelských dotazů zadávaných do vyhledávače. Předpokládá se, že v korpusu je na každém řádku uložen dotaz ve formátu: časový_údaj<tabulátor>dotaz. Časový údaj nevyužíváme.

Plnit jazykový model přímo z textového korpusu je velmi časově náročné, a je lepší používat metody: load_from_modelfiles a save_models.

`save_models(self, path_uni, path_bi, path_ibi, encoding="utf-8")`

Obalová metoda pro metody save_bigrams a save_unigrams.

`save_unigrams(self, path, encoding="utf-8")`

Metoda uloží unigramový model do souboru kvůli rychlejšímu nahrávaní. Na jednotlivých řádcích bude v souboru uloženo: slovo<tab>četnost.

`save_bigrams(self, path, inv, encoding="utf-8")`

Metoda uloží bigramový model do souboru kvůli rychlejšímu nahrávaní. Parametr inv určuje, zda budeme ukládat inverzní, nebo normální bigramový model. Na jednotlivých řádcích v souboru bude uloženo:

první_slovo<tab>následující_slovo<tab>četnost.

`load_from_modelfiles(self, path_uni, path_bi, path_ibi, encoding="utf-8")`

Obalová metoda pro metody load_bigrams a load_unigrams.

`load_unigrams(self, path, encoding="utf-8")`

Metoda naplní unigramový model ze souboru ve formátu, v jakém jej uložila metoda save_unigrams.

`load_bigrams(self, path, inv, encoding="utf-8")`

Metoda naplní bigramový model ze souboru ve formátu, v jakém jej uložila metoda save_bigrams. Parametr inv určuje, zda budeme nahrávat do inverzního bigramového modelu, nebo do normálního.

`score(self, contex_word, word, first)`

Obalová metoda pro unigram_probability a bigram_probability. Pokud není zadáno contex_word, bude se počítat dle unigram_probability, v opačném případě dle bigram_probability. Parametr first určuje, zda je word první v dotazu a jestli se bude počítat pravděpodobnost podle inverzního, nebo normálního bigramového modelu.

`unigram_probability(self, word)`

Spočítá relativní čestnost slova word. Pokud se slovo v modelu nenachází, je použito vyhlazování metodou add lambda.

`bigram_probability(self, contex_word, word, inv)`

Spočítá podmíněnou pravděpodobnost dvojici slov contex_word a word.

Jestliže je hodnota parametru inv True, spočítá pravděpodobnost, že slovo word předchází slovu contex_word. Pokud je hodnota False, bude počítat pravděpodobnost, že slovo word následuje za slovem contex_word.

Pokud se slova v modelu nenacházejí, je použito vyhlazování metodou add lambda.

`edits1(self, word)`

Metoda vrátí množinu (set) všech možných variant slova word po aplikaci editačních operací do vzdálenosti 1. Editační operace využívají abecedu CZ_APLHABETH.

`known(self, words)`

Ze zadaného seznamu slov words vrátí množinu (set) pouze těch slov, která se vyskytují v unigramovém modelu.

`known_edits2(self, word)`

Metoda vrátí množinu (set) všech možných variant slova word po aplikaci editačních operací pro vzdálenost 2 a pouze těch, které se vyskytují v unigramovém modelu. Editační operace využívají abecedu CZ_APLHABETH.

`known_edits2_weight(self, contex_word, word, lvl, first)`

Metoda vrátí slovník ohodnocených možných variant slova po aplikaci editačních operací pro vzdálenost 2 a pouze těch, které se vyskytují v unigramovém modelu. Ohodnocení probíhá podle jazykového i podle chybového modelu. Úroveň přesnosti ohodnocení určuje parametr lvl. Editační operace využívají abecedu CZ_APLHABETH.

`known_edits1_weight(self, contex_word, word, lvl, first)`

Obalová metoda pro metody transpone, subtitute, delete, insert. Tyto metody vrátí slovník ohodnocených možných variant slova po aplikaci editačních operací do editační vzdálenosti 1 a pouze těch, které se vyskytují v unigramovém modelu. Ohodnocení probíhá podle jazykového i podle chybového modelu. Úroveň přesnosti ohodnocení určuje parametr lvl.

`unigram_entropy(self)`

Spočítá entropii unigramového modelu.

`bigram_entropy(self, inv)`

Spočítá entropii bigramového modelu.

4.3.2 error_model

Modul je určen pro práci s chybovým modelem. Obsahuje jedinou třídu ErrorModel

ErrorModel

Třída uchovává různé chybové statistiky, podle kterých bude hodnotit kandidáty. Tyto statistiky jsou uloženy v seznamu (list) errors. Ten obsahuje pro jednotlivé editační vzdálenosti slovníky s podrobnějšími údaji.

Mezi nejdůležitější funkce modelu patří ohodnocování kandidátů, ke které se používá metoda score.

`__init__(self, dir)`

Parametr dir obsahuje cestu k adresáři obsahujícímu soubory s chybovými statistikami.

`load_error_stats(self, dir)`

Nahraje soubory s chybovými statistikami z adresáře dir do modelu. Soubory musí splňovat určité jmenné konvence.

Statistiky pro editační vzdálenosti – errors_dist.txt

Statistiky jednotlivé operace – errors_oper.txt

Statistiky pro znaky – <oper>_errors.txt, kde <oper> je zkratka operace

`score(self, edit, oper, char, lvl)`

Metoda spočítá ohodnocení chybového modelu dle zadaných parametrů. Edit je editační vzdálenost, oper je použitá operace, char je ovlivněný znak a lvl je úroveň přesnosti.

Pro první úroveň ohodnocení se bere v úvahu pouze editační vzdálenost.

Na druhé úrovni se započítává i použitá operace.

Pro třetí úroveň se bere v potaz i použitý znak.

Ještě poznamenáme, že po úroveň vyšší než 1 a pro editační vzdálenosti větší než 1 se hodnoty kvůli vysoké výpočetní náročnosti pouze aproximují ze statistik pro editační vzdálenost 1.

4.3.3 error_analysis

Modul se používá pro chybovou analýzu dotazů. Obsahuje třídu ErrorAnalyser.

ErrorAnalyser

Třída slouží k analýze a generování chybových statistik, které se využívají v chybovém modelu. Pro tvorbu chybových statistik byl k dispozici záznam dotazů obsahujících překlepy a jejich opravy. Každá dvojice dotazů byla v souboru umístěna na samostatném řádku a oddělena tabulátorem.

Třída vlastní dva slovníky (dict): errors a corrections. Slovník corrections obsahuje slova a jejich opravy. V errors budou uloženy záznamy o chybách.

`create_error_file(self, path_in, path_out, encoding="utf-8")`

Ze souboru se záznamy dotazů s překlepy vybere pouze chybná slova a jejich korektní protějšek. Následně je uloží do výstupního souboru odděleného tabulátorem.

`load_error_file(self, path, encoding="utf-8")`

Nahraje soubor dvojic vytvořené metodou create_error_file do slovníku corrections.

`errors_from_alignment(self, correct, wrong, dl=True)`

Zarovná slova correct a wrong. Ze zarovnání následně určí editační vzdálenost, seznam použitých editačních operací i s parametry a jejich pozici ve slově. Tyto údaje uloží do slovníku a vrátí jej.

`analyse_errors(self, encoding="utf-8", dl=True)`

Provádí kompletní chybovou analýzu. Postupně prochází chybové dvojice v corrections, aplikuje na ně metodu errors_from_alignment a její výsledky ukládá do slovníku errors.

`analyse_edit_dist(self, dist_limit=5)`

Na základě údajů ve slovníku errors provede chybovou analýzu editačních vzdáleností.

Vrátí slovník obsahující pravděpodobnosti zastoupení editačních vzdáleností v chybovém souboru.

`analyse_error_opers(self, distance=1)`

Na základě údajů ve slovníku errors provede pro zadanou editační vzdálenost distance chybovou analýzu editačních operací.

Vrátí slovník obsahující pravděpodobnosti zastoupení distančních operací v chybovém souboru.

`analyse_chars(self, oper, distance=1)`

Metoda pro zadanou vzdálenost distance a editační operaci oper provede chybovou analýzu ovlivněných znaků.

Vrátí slovník obsahující pravděpodobnosti zastoupení ovlivněných znaků v chybovém souboru.

4.3.4 edit_distance

Modul obsahuje rozmanité funkce, které operují s editační vzdáleností. Parametr dl u všech funkcí určuje, zda budeme počítat Damerau-levenshteinovu (True), nebo levenshteinovu vzdálenost (False).

`distance_matrix(word, similar, dl)`

Pro zadaná slova word a similar vygeneruje vzdálenostní matici, ze které se počítá editační vzdálenost. K uložení této matice se používá seznam (list). Použitý algoritmus spadá do třídy dynamického programování.

`print_dist_matrix(word, similar, dl)`

Pro zadaná slova word a similar vytiskne hezky zformátovanou vzdálenostní matici na standardní výstup.

`distance(word, similar, dl)`

Pro zadaná slova word a similar spočítá editační vzdálenost.

`alignment(word, similar, dl)`

Pomocí vzdálenostní matice zarovná a vrátí zadaná slova word a similar.

4.3.5 utils

Modul obsahuje různé pomocné funkce, které nezapadají do jiných modulů.

`undiacritic(text, encoding="utf-8")`

Ze zadaného řetězce text odstraní diakritiku.

`normalize_file(path_r, path_w, stack_size=50, lower=True)`

Tato funkce se používá k normalizaci textového korpusu. Pomocí zásobníku odstraní ze souboru krátce po sobě se opakující duplicitní řádky. Rovněž nabízí převod na malá slova.

`calculate_entropy(freq_i_j, freq_i)`

Spočítá entropii pro dané pravděpodobnosti.

4.3.6 corrector

Modul obstarává korekci slov. Obsahuje třídu Corrector.

Poznámka: korekční algoritmus nedodržuje pořadí fází tak, jak bylo uvedeno v teoretické části diplomové práce. Kvůli optimalizacím jsou kandidáti ohodnocovaní chybovým i jazykovým modelem zároveň při generování.

Corrector

Stěžejní třída celého korektoru, které opravuje uživatelské dotazy.

`__init__(self, lang_model)`

Jediným parametrem konstruktoru je jazykový model, který už v sobě obsahuje chybový model.

`correct_query(self, query, lvl, min_len=1)`

Rozdělí zadaný dotaz query podle bílých znaků na posloupnost slov, které pak předává metodě correct. Jakmile jsou všechna slova opravena, složí z nich zpět dotaz a ten vrátí.

Metoda zohledňuje problém kontextu prvního slova v dotazu tak, jak byl popsán v teoretické části. Při opravě prvního slova nemůžeme využít kontext předcházejícího slova, proto využijeme kontext následujícího.

Parametr lvl určuje přesnost ohodnocení dle chybového modelu. Detaily lze nalézt v modulu error_model. Pokud je hodnota lvl 0 slovo, bude opraveno pomocí correct_naive.

Parametr min_len určuje minimální délku slova, které se bude opravovat. Pokud je slovo kratší než tato vzdálenost, bude přeskočeno.

`correct(self, contex_word, word, lvl, first)`

Metoda opraví slovo word na základě kontextu slova contex_word.

Pomocí jazykového modelu vygeneruje ohodnocené kandidáty do vzdálenosti 2. Z kandidátů vybere a vrátí toho s největším ohodnocením.

Pokud je parametr first True, bude metoda zohledňovat problém kontextu prvního slova v dotazu.

Parametr lvl určuje přesnost ohodnocení dle chybového modelu. Pokud je hodnota lvl 0 slovo, bude opraveno pomocí correct_naive.

`correct_naive(self, contex_word, word, first)`

Metoda opraví slovo pomocí naivní metody tak, jak je popsáno v teoretické části diplomové práce.

4.4 Skripty

Na začátek připomeneme, že všechny skripty vyžadují korektně nainstalovaný balíček **spell-correction**. Většina těchto skriptů má pouze pomocný jednoúčelový charakter a jejich programátorská čistota je nevalná. Nejedná se o dávkové soubory a musí být spuštěny jako parametry interpretu jazyka Python. Jednotlivé skripty se nacházejí v adresáři:

`/src/`

a budou se spouštět:

```
python jmeno_skriptu.py
```

Různé cesty k souborům a parametry jsou pevnou součástí zdrojového kódu. Proto nechceme-li měnit zdrojový kód, je pro bezproblémový chod nezbytné dodržet rozvrženou adresářovou strukturu.

Skripty úzce souvisí s dílčími úlohami opravy překlepu, proto si je popíšeme v kontextu řešených problémů.

4.5 Příprava jazykového modelu

Pro tvorbu jazykového modelu byl použit korpus sestavený ze skutečných dotazů, které uživatelé zadávají do vyhledávače společnosti Seznam.cz. Korpus *korpus.test.9mil.txt* měl velikost 268MB a obsahoval deset miliónu dotazů. Každý dotaz v textovém korpusu byl vždy umístěn na samostatném řádku spolu s časovým údajem. V tomto korpusu se řada stejných dotazů opakovala krátce po sobě, což mohlo být zapříčiněno například nekvalitním internetovým připojením uživatele.

S využitím skriptu *normalize_korpus.py* byly odstraněny krátce se po sobě opakující duplicity. Skript využívá funkci *normalize_text* z modulu **utils**. Velikost použitého zásobníku byla 900 položek. Veškerý text byl zároveň převeden na malá písmena. Ve znormalizovaném korpusu *queries-search.timestamps.norm.txt* zbylo asi devět miliónů položek a jeho velikost byla 250MB.

Ze základního korpusu se vytvořily tři menší, aby bylo možné testovat závislost úspěšnosti na objemu dat. Korpusy byly získány použitím unixového příkazu *head* a jejich velikosti jsou: 3, 6 a 9 miliónů dotazů. Korpusy jsou pojmenovány podle schématu: *korpus.test.[velikost]mil.txt*.

Nyní musíme z těchto korpusů sestavit unigramové a bigramové modely. Tvorba modelů přímo z korpusu je časově náročná, proto si musíme modelové soubory připravit dopředu. K tomuto účelu se využívá skript *build_language_model.py* postavený na modulu **language_model**. Vytvořené soubory s modely jsou uloženy v adresáři:

```
/data/models/language/
```

Soubory s unigramovými modely mají název *unigrams.[velikost]mil.[min_delka].txt* a s bigramovými *bigrams.[velikost]mil.[min_delka].txt*. Parametr *min_delka* označuje minimální délku slova v modelu.

4.6 Příprava chybového modelu

Pro tvorbu chybových statistik byl k dispozici soubor dotazů *queries-error.txt* obsahující překlepy a jejich opravy. Každá dvojice dotazů byla v souboru umístěna na samostatném řádku a oddělena tabulátorem. Soubor byl opět poskytnut společností Seznam.cz a získán z reálného provozu.

Pomocí skriptu *build_error_file.py* z tohoto souboru vybereme pouze chybná slova a jejich korektní protějšek a uložíme je do výstupního souboru *errors.txt* oddělené tabulátorem. Soubor *errors.txt* obsahuje přes devadesát tisíc položek. Z tohoto souboru pak skriptem *build_error_model.py* vygenerujeme detailní statistiky, které budou použity v chybovém modelu.

4.7 Příprava testů korekce

Korektor se testuje ze všech možných hledisek pomocí skriptu *correction_tests_final.py*. Více si o samotném testování řekneme v následující kapitole.

4.8 Demonstrační skript

Mezi skripty nalezneme i jednoduchý korektor spustitelný z příkazové řádky. Korektor se spouští skriptem *correct_me.py* a ukončuje zadáním dotazu __exit. Opravy probíhají v interaktivním režimu.

5 Testování

Sadou testů vytvořených na základě podkladů z reálného provozu prověříme, zda se nám podařilo vypořádat se všemi nástrahami korekce překlepů pro vyhledávač.

Konfigurace použitého počítače byla: Intel Core Duo 1,75 GHz se 3 GB operační paměti. Na této konfiguraci trval běh celé testovací sady asi 8 hodin.

5.1 Vstupní data jazykového modelu

Pro účely testování byly vytvořeny celkem tři velikosti jazykových modelů (o 3, 6, 9 miliónech dotazů) ve dvou variantách, které se liší minimální délkou slova v modelu. Během dílčího testování, kdy modely neobsahovaly pouze jednoznaková slov, se ukázalo, že řada krátkých slov bývá opravována na předložky: *na*, *ke*, *do*. Proto jsme zkusili vytvořit druhou variantu modelu, který neobsahuje ani dvouznaková slova, a budeme sledovat, jak se tato úprava projeví na přesnosti oprav.

Do tabulky 3 jsme shrnuli důležité parametry jednotlivých modelů. Tabulka 4 obsahuje seznam nejčastějších unigramů a částečně také napovídá, proč docházelo k nežádoucím opravám krátkých slov.

Tabulka 3. Parametry jazykových modelů

Velikost korpusu	3 mil.		6 mil.		9 mil.	
Minimální délka slov	2	3	2	3	2	3
Velikost slovníku (tis)	368	366	591	589	784	782
Počet unigramů (tis)	6 674	6 205	13 419	1 246	20 607	19 143
Počet bigramů (tis)	3 677	3 215	7 426	6 487	11 392	9 951
Entropie	5,98	5,46	6,86	6,14	8,14	7,58

5.2 Vstupní data chybového modelu

Chybové statistiky dotazů zadávaných do vyhledávače posloužily jako základ pro tvorbu chybového modelu. S některými údaji se krátce seznámíme.

V tabulce 5 jsou uvedeny procentuální zastoupení chyb podle editační vzdálenosti. Z těchto údajů je patrné, že v dotazech výrazně převažují slova obsahující jeden překlep.

Podrobnější statistiky jsou k dispozici pouze pro chyby s editační vzdáleností 1. Podle tabulky 6 mezi nejčastější editační operace patří záměna znaků a za ní následuje

Tabulka 4. Nejčastější unigramy podle minimální délky

Minimální délka	
2	3
na	mp3
mp3	zdarma
zdarma	hry
hry	download
ke	pro
download	stažení
pro	online
do	stazeni

vložení. Výskyt transpozice je zanedbatelný.

Z tabulky ovlivněných znaků 7 je vidět, že řadě uživatelů působí problémy psaní y a i .

Tabulka 5. Pravděpodobnostní rozdělení editačních vzdáleností

Vzdálenost	Pravděpodobnost
1	0,72
2	0,12
3	0,07
4	0,07

Tabulka 6. Pravděpodobnostní rozdělení editačních operací pro vzdálenost 1

Editační operace	Pravděpodobnost
záměna	0,50
vložení	0,29
smazání	0,19
transpozice	0,008

5.3 Testy

Všechny testy byly vytvořeny na základě skutečných dat získaných ze záznamů dotazů společnosti Seznam.cz.

Tabulka 7. Pravděpodobnostní rozdělení ovlivněných znaků

Vložení		Záměna		Smazání		Transpozice	
Znaky	Prav.	Znaky	Prav.	Znaky	Prav.	Znaky	Prav.
e	0,093	y → i	0,056	s	0,089	l → a	0,035
a	0,086	i → y	0,043	e	0,087	a → r	0,030
s	0,084	e → a	0,033	a	0,076	r → a	0,027
n	0,066	a → e	0,029	y	0,074	l → e	0,024
o	0,061	a → y	0,026	n	0,064	n → a	0,023

Každý test se skládá z posloupnosti dvojic obsahujících chybný tvar a očekávaný tvar. Při procházení testu dostane korektor na vstup chybný tvar a na výstupu vrátí navrhovanou opravu, která se porovná s očekávaným tvarem. Při porovnání se nebere v úvahu velikost znaků a diakritika. Slova *Oldřich* a *oldrich* jsou pro naše účely totožné. Na závěr testu se vyhodnotí procentuální úspěšnost, kolikrát se výstup korektoru shodoval s očekávaným slovem. Rovněž se měří celková doba vykonávání testu. Celkem máme připraveny tři testy, které prověří korektor z různých hledisek.

Testy budeme provádět s různým počátečním nastavením korektoru, které obnáší různé velikosti a varianty jazykového modelu a úrovně přesnosti chybového modelu.

Všechny testy jsou k dispozici na CD příloze.

5.3.1 Test izolovaných slov

isolated.txt (235 položek) Test izolovaných slov prověřuje schopnost korektoru poradit si s překlepy bez kontextu. Tento test je tedy složený pouze z jednoslovných chybných dotazů.

Tabulka 8. Výsledky testu izolovaných slov pro 3 mil.

Minimální délka slov	1				2			
Úroveň přesnosti	0	1	2	3	0	1	2	3
Úspěšnost (%)	20	49	46	22	20	49	45	22
Čas (s)	26	284	279	273	28	270	317	283

Tabulka 9. Výsledky testu izolovaných slov pro 6 mil.

Minimální délka slov	1				2			
Úroveň přesnosti	0	1	2	3	0	1	2	3
Úspěšnost (%)	19	52	48	25	19	52	47	25
Čas (s)	21	298	298	303	21	291	293	283

Tabulka 10. Výsledky testu izolovaných slov pro 9 mil.

Minimální délka slov	1				2			
Úroveň přesnosti	0	1	2	3	0	1	2	3
Úspěšnost (%)	19	54	50	28	19	54	49	26
Čas (s)	21	288	303	311	23	297	321	306

5.3.2 Test běžných dotazů

normal.txt (262 položek)

Test je sestaven z častých dotazů obsahujících překlepy a věrně simuluje chyby, se kterými se korektor setká při běžném provozu. Důsledně se tak prověří schopnosti korektoru, ze všech různých stránek. V tomto testu bude moci korektor využít k opravě kontext okolních slov. Výsledky tohoto testu nám dávají nejrelevantnější výpověď o kvalitě korektoru.

Tabulka 11. Výsledky testu běžných dotazů pro 3 mil.

Minimální délka slov	1				2			
Úroveň přesnosti	0	1	2	3	0	1	2	3
Úspěšnost (%)	20	42	32	12	17	38	29	11
Čas (s)	42	622	622	623	43	632	614	620

5.3.3 Test konzervativnosti

konservative.txt (765 položek) Důležitou vlastností korektoru je konzervativnost. Korektor by se neměl snažit opravit dotaz za každou cenu, a pokud si s nim neví rady, měl by jej nechat v původním tvaru. Test konzervativnosti obsahuje jen 7% překlepů, zbylé dotazy jsou správné. Tento test poměrně věrně simuluje skutečný provoz na vyhledávači, kdy převážnou část dotazů není potřeba opravovat. Sám o sobě nám však

Tabulka 12. Výsledky testu běžných dotazů pro 6 mil.

Minimální délka slov	1				2			
Úroveň přesnosti	0	1	2	3	0	1	2	3
Úspěšnost (%)	19	42	36	15	14	38	32	13
Čas (s)	63	638	673	653	34	656	648	659

Tabulka 13. Výsledky testu běžných dotazů pro 9 mil.

Minimální délka slov	1				2			
Úroveň přesnosti	0	1	2	3	0	1	2	3
Úspěšnost (%)	19	43	38	17	19	38	35	14
Čas (s)	71	703	712	743	78	703	743	703

tento test mnoho o schopnostech korektoru neprozradí, a proto jej ilustrativně spustíme pouze s největším jazykovým modelem. Je zřejmé, že by v něm dosáhl velké úspěšnosti i korektor, který by neopravil žádný dotaz. Test byl připraven společností Seznam.cz

Tabulka 14. Výsledky testu konzervativnosti pro 6 mil.

Minimální délka slov	1				2			
Úroveň přesnosti	0	1	2	3	0	1	2	3
Úspěšnost (%)	80	89	91	94	73	89	92	94
Čas (s)	489	2288	2010	2292	489	2020	2282	2013

6 Závěr

V rámci diplomové práce se podařilo proniknout do problematiky korekce překlepů včetně přidružených disciplín. Všechny získané poznatky byly podrobně sepsány. Práce tak tvoří jeden z nejucelenějších česky psaných zdrojů věnovaný problematice oprav překlepů.

Na základě získaných znalostí byl navržen korekční algoritmus a implementován prototyp. Na první pohled není výkon prototypu korektoru příliš přesvědčivý. Průměrná doba opravy jednoho běžného dotazu skládajícího se ze tří slov se pohybuje mezi jednou a dvěma sekundami. Musíme však vzít v úvahu, že prototyp byl implementován v interpretovaném jazyce a pro uložení jazykových a chybových modelů byly použity obecné datové struktury. Zvýšení výkonu oprav bychom mohli snadno dosáhnout reimplementací algoritmu v některém z kompilovaných jazyků a použitím sofistikovaných datových struktur. Z tohoto pohledu si algoritmus vzhledem k použitým prostředkům nestojí špatně.

Jedním z požadavků na algoritmus bylo, aby při opravách uměl využít kontextu okolních slov. K uložení a ohodnocení kontextu využíváme statistických metod a dvojice slov ukládáme do jazykového bigramového modelu. Vzhledem ke krátké historii slov v dotazu by použití vyšších řádů $n-$gramových modelů nemělo smysl. K vyhlazování jsme kvůli výkonu použili metodu add lamda.

Největší překážka v korekci dotazů do vyhledávače je jejich specifický jazyk. S tímto problémem jsme se vypořádali zvolením obecné metriky pro popis překlepů, kterou je minimální ediční vzdálenost. Použitá metoda je však nedostačující pro korekci určitého druhu chyb.

Problematické jsou ty typy chyb, jejichž důsledkem je rozdělení, nebo spojení slova. Pro současnou verzi algoritmu jsou nerozlousknutelným oříškem, ale naštěstí se s nimi nesetkává příliš často.

Dalším dosud nevyřešeným problémem je česká fonetická transkripce anglických slov (*gugl*, *noutbuk*, *kul*), která má často velkou editační vzdálenosti. Dala by se však odstranit použitím speciálního slovníku.

Algoritmus umí opravovat pouze slova do editační vzdálenosti 2, ale z chybových statistik vyplynulo, že až 7% chyb má vzdálenost 3. Počet kandidátů pro vzdálenost tři by byl astronomický, ale bylo by možné zaměřit se na určitou problematickou podmnožinu a z té kandidáty generovat.

Celkové výsledky z provedených měření by se daly označit za uspokojivé. Při nej-

lepším nastavení algoritmus opraví zhruba 40% dotazů, ale touto úspěšností se nemůže rovnat komerčním korektorům velkých firem, které dosahují úspěšnosti přes 70%.

Z testů rovněž vyplynulo několik zajímavých skutečností. Bezkonkurenčně nejrychlejší se ukázala být naivní metoda korekce, nebo také nultá úroveň přesnosti chybového modelu. V testech bývá 5-10krát rychlejší než ostatní metody a přitom dosahuje o 10-15% nižší úspěšnosti. Poměrně překvapující je fakt, že s vyšší úrovní přesnosti klesá úspěšnost oprav. Na vinně může být několik faktorů. Statistické rozložení chyb v testu nemusí odpovídat hodnotám chybového modelu. Dále je možné, že i přes velký objem trénovacích dat nebylo dosaženo věrohodných statistických hodnot. Za zkreslení může částečně také fakt, že ohodnocení pro editační vzdálenost 2 je aproximováno.

Nejvyšší úspěšnosti jsme dosáhli na první úrovni přesnosti, kdy kandidáty hodnotíme pouze podle editační vzdálenosti. Naopak, jako nejméně úspěšná se ukazuje třetí úroveň přesnosti, která se se svým výsledkem blíží k naivní metode.

Objemnější jazykový model má pozitivní vliv na úspěšnost oprav. Tato závislost ovšem není lineární, ale spíše logaritmická a při určitém objemu bychom narazili na hranici, kdy bude nárůst úspěšnosti zanedbatelný.

Při testování se nepotvrdila myšlenka, že omezením minimální délky slova v modelu dosáhneme větší přesnosti. Omezené jazykové modely dosahovaly horších výsledků.

Úspěšnosti oprav by mohlo znatelně pomoci použití pokročilejší metody vyhlazování jazykového modelu. Pro implementaci byla z výkonnostních důvodů použita nepříliš účinná metoda add lambda. Spolu s implementací v kompilovaném jazyce by mohla přibýt i podpora pro sofistikované vyhlazovací metody.

Podle původního zadání se očekávalo, že v diplomové práci bude využito korpusové lingvistiky. Po prostudování problematiky se ukázalo použití této disciplíny jako nevhodné. Korpusová lingvistika je založena na myšlence, že pokud máme k dispozici dostatečně rozsáhlý a kvalitní soubor textu, dokážeme na něm s velkou přesností pozorovat různé jazykové jevy a získané závěry budou platit pro širokou oblast zkoumaného jazyka. Pro korpusovou lingvistiku je tedy korpus kvalitativní prostředek pro jazykovědný výzkum, zatímco statistická lingvistika k němu přistupuje pouze jako ke kvantitativnímu souboru statistických dat.

Reference

[1] J. Černỳ. Úvod do studia jazyka. *Olomouc: Rubico*, 1998.

[2] Psutka, J.; Müller, L.; Matoušek, J.; Radová, V. *Mluvíme s počítačem česky.* Academia, Prague, 2006. Dostupné z: `http://www.kky.zcu.cz/en/publications/PsutkaJ_2006_Mluvimes`.

[3] D. Jurafsky and J.H. Martin. Speech and Language Processing: An Introduction to Natural Language Processing. *Computational Linguistics, and Speech Recognition, Prentice Hall PTR, Upper Saddle River, NJ*, 2000.

[4] K. Kukich. Technique for automatically correcting words in text. *ACM Computing Surveys (CSUR)*, 24(4):377–439, 1992.

[5] S. Cucerzan and E. Brill. Spelling correction as an iterative process that exploits the collective knowledge of web users. *Proceedings of EMNLP*, 4:293–300, 2004.

[6] Autor nezmámý. *Test of Phonetic Lookup Using Metaphone*, 1999. [Online]. Dostupné z: `http://www.lanw.com/java/phonetic/default.htm`.

[7] A.R. Golding and D. Roth. A Winnow-Based Approach to Context-Sensitive Spelling Correction. *Machine Learning*, 34(1):107–130, 1999.

[8] G. Ward. *Installing Python Modules*, 2008. [Online]. Dostupné z: `http://docs.python.org/inst/inst.html`.

[9] P. Norvig. *How to Write a Spelling Corrector*, 2006. [Online]. Dostupné z: `http://www.norvig.com/spell-correct.html`.

[10] T. White. *Can't beat Jazzy*, 2004. [Online]. Dostupné z: `http://www.ibm.com/developerworks/java/library/j-jazzy/`.

[11] A.R. Golding. A Bayesian hybrid method for context-sensitive spelling correction. *Proceedings of the Third Workshop on Very Large Corpora*, 3:39–53, 1995.

[12] E. Mays, F.J. Damerau, and R.L. Mercer. Context based spelling correction. *Information Processing and Management: an International Journal*, 27(5):517–522, 1991.

[13] M.D. Kemighan, K.W. Church, and W.A. Gale. A Spelling Correction Program Based on a Noisy Channel Model. *Proc. of COLING-90*, pages 205–2, 1990.

[14] F.J. Damerau. *A Technique for Computer Detection and Correction of Spelling Errors*, 1963.

[15] F. Ahmad and G. Kondrak. Learning a spelling error model from search query logs. *Proceedings of EMNLP 2005*, pages 955–962, 2005.

[16] E. Brill and R.C. Moore. An improved error model for noisy channel spelling correction. *Proceedings of the 38th Annual Meeting on Association for Computational Linguistics*, pages 286–293, 2000.

[17] D. Harms. *McDonald K.: Začínáme programovat v jazyce Python*, 2003.

[18] M. Lutz and D. Ascher. *Naučte se Python.* Grada, 2003.